## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 1: Daten lesen und verändern

**Als Voraussetzung für das Programmieren in Maschinensprache bietet der ZX 81 zwei Basic-Befehle, die den Zugriff auf einzelne Bytes im Speicher erlauben.**

Noch vor wenigen Jahren waren sogenannte Heimcomputer so beschaffen, daß man beim Programmieren nicht um Maschinensprache, also um die Befehlseingabe in hexadezimaler Form, herumgekommen ist. Moderne Heimcomputer verstehen hingegen Basic und können ohne jede Kenntnis von Maschinensprache gut programmiert werden. Aber die Maschinensprache lebt trotzdem weiter. Das zeigt sich bei einem Blick in Mikrocomputer-Zeitschriften oder beim Auflisten professionell geschriebener Programme und hat gute Gründe.

Als Einsteiger freilich steht man dem Maschinencode ratlos gegenüber, und die Bedeutung von Programmzeilen mit seltsamen REM-Kommentaren bleibt im Dunkeln. Wer sich aktiv mit seinem Heimcomputer beschäftigt und dessen Leistungsfähigkeit ausschöpfen möchte, wird aber gewiß nicht lange in der „Schreckstarre" verharren, sondern den Dingen auf den Grund gehen. Dabei will die Serie „Klartext für den ZX 81" Schrittmacherdienste leisten und leicht verständlich mit vielen praktischen Übungen das Programmieren in Maschinensprache verständlich machen. Da besonders Einsteiger angesprochen sind, wurde der ZX 81 als Übungsmodell ausgewählt.

### Maschinenprogramme sorgen für Tempo

Maschinensprache ist eine elementare Sprache, auf die andere, höhere Programmiersprachen aufbauen. So wurde von den Entwicklern des ZX 81 erst einmal ein Maschinenprogramm geschrieben, das dem Computer die Programmiersprache Basic verständlich macht. Es wurde im ROM (Lesespeicher) des ZX 81 verankert und kann nicht verändert werden. Wenn Sie also ein Basic-Programm eingegeben haben und es durch RUN zum Laufen bringen, wird jedes Zeichen, jedes Schlüsselwort vom ROM übersetzt, d. h. interpretiert: Man nennt das Maschinenprogramm im ROM deshalb „Interpreter".

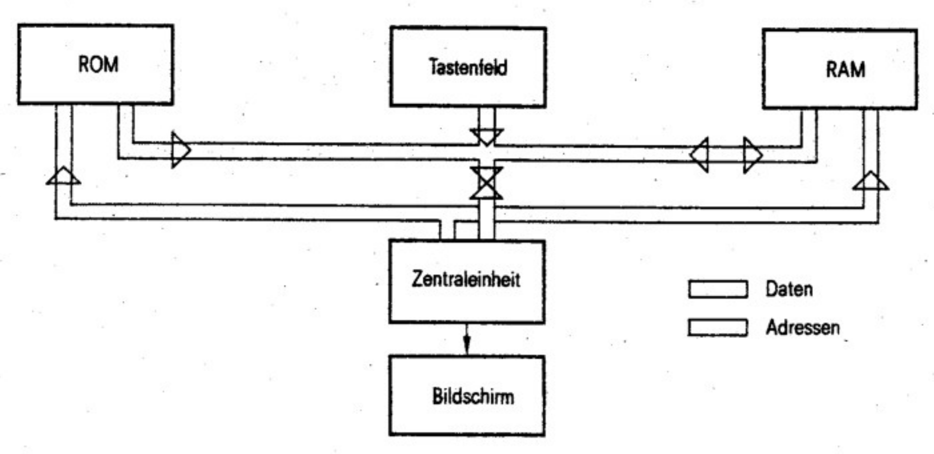
Der Interpreter macht der Zentraleinheit im Computer verständlich, was die Basic-Programmschritte bezwekken sollen. Leider nimmt dieses andauernde Interpretieren viel Zeit in Anspruch, was sich besonders bei schnellen Spielen, bei bewegten Grafiken, beim Rechnen und überall dort, wo Zeit kostbar ist, störend bemerkbar macht. Es muß also gelingen, den Interpreter zu übergehen und gleich ein für den Computer unmittelbar verständliches Maschinenprogramm anstelle des Basic-Programms zu schreiben. Doch bevor wir damit beginnen können, ist etwas Theorie notwendig.

### Die Hausnummern der RAM-Straße

Neben dem ROM hat der ZX 81 auch noch ein RAM (Arbeitsspeicher). Wie *Bild 1* zeigt, lassen sich hier im Gegensatz zum ROM die Daten auch verändern. Selbstgeschriebene Maschinenprogramme können also nur dort abgelegt werden. Doch wie sieht nun so ein Speicher aus?

Ein Speicher besteht aus vielen einzelnen Zellen, in die jeweils ein Bit paßt. Damit man sie gezielt ansprechen kann und nicht verwechselt, werden jeweils 8 Bit zu einer Gruppe zusammengefaßt (1 Byte) und mit einer Art Hausnummer versehen, nämlich einer Adresse. Ein gespeichertes Programm ist in solchen 8-Bit-Speicherzellen untergebracht. Sehr wichtig ist, daß jedes Byte dann eine Dezimalzahl von 0 bis 255 darstellen kann ($255 = 2^8 - 1$; Eins wird abgezogen, weil die Zählung bereits mit Null beginnt).

Betrachten wir jetzt den Aufbau des Arbeitsspeichers (RAM) näher: Sein Bereich beginnt, wie aus *Bild 2* ersichtlich, bei Adresse 16 384. Ohne Zusatzspeicher reicht er 1024 Byte (1 KByte)




① **Blockschaltung eines Computers:** Die Zentraleinheit kann mit 16-Bit-Adressen max. 65 535 8-Bit-Speicherzellen im RAM adressieren

weit bis zur Adresse 17 407. Mit der 16-KByte-Erweiterung reicht der Arbeitsspeicher bis zur Adresse 32 767.

Jede Adresse wird nun beim ZX 81 (und bei vielen anderen Computern) durch zwei Bytes dargestellt, wobei die größte Dezimalzahl, die mit zwei Bytes erreicht werden kann, die Zahl 65 535 ist. Darauf kommt man, wenn man 256 quadriert und vom Ergebnis Eins abzieht (oder: $65\,535 = 2^{16} - 1$; der Exponent macht deutlich, daß der ZX 81 16 Adreßleitungen hat).

Doch zurück zum Arbeitsspeicher: Bei Adresse 16 384 beginnt der erste Teilbereich, der Bereich mit den Systemvariablen. Diese wollen wir vorerst beiseite lassen. Interessant wird es bei Adresse 16 509, denn dort beginnt der Bereich des Programmspeichers.

## Ein Basic-Befehl deckt Speicherzellen auf

Die Sache mit der Adressierung von Speicherzellen wollen wir jetzt gleich in der Praxis erproben. Es ist daher Zeit geworden, den ZX 81 einzuschalten. Geben Sie danach ein:

10 REM ABCDEF

Überlegen wir uns einmal, welche Adresse das „A" haben müßte. Wie gezeigt beginnt der Programmspeicher bei Adresse 16 509. Die ersten zwei Bytes werden für die Zeilennummer, die folgenden beiden Bytes für die Länge der Programmzeile beansprucht. Danach folgt das Byte für die REM-Anweisung. Erst dann, unter der Adresse 16 514, folgt der Code für das „A". In *Bild 3* wird dieser Sachverhalt verdeutlicht.

Ob das „A" tatsächlich unter der angegebenen Adresse gespeichert ist, kann jeder selbst durch Anwenden des PEEK-Befehls nachprüfen. Durch PEEK n erfährt man nämlich vom ZX 81 den Wert des Bytes, welches unter der Adresse n steht. Geben Sie jetzt ein:

PRINT PEEK 16514

Der Computer muß jetzt den Code von „A", nämlich 38 anzeigen (siehe Anhang A des ZX-81-Programmierhandbuchs von Sinclair). Selbstverständlich lassen sich auf diese Weise auch die übrigen Adressen abfragen.

## Wie durch Geisterhand: die REM-Zeile wird verändert

Es bringt einen natürlich nicht viel weiter, wenn man Inhalte von Adressen lediglich abfragen kann (in der Fachsprache heißt das „peeken"). Vielmehr wollen wir die Inhalte auch verändern. Dafür gibt es in Basic den POKE-Befehl. Durch POKE n,m kann man das Byte unter der Adresse n auf die Zahl m setzen.

Versuchen wir also, aus dem „A" in der REM-Zeile ein „G" zu machen. Die Adresse von „A" ist noch immer 16 514. Deshalb darf man getrost
POKE 16514,CODE "G"
oder
POKE 16514,44
eingeben. Nach dem Auflisten wird die REM-Zeile die erwünschte Veränderung zeigen.

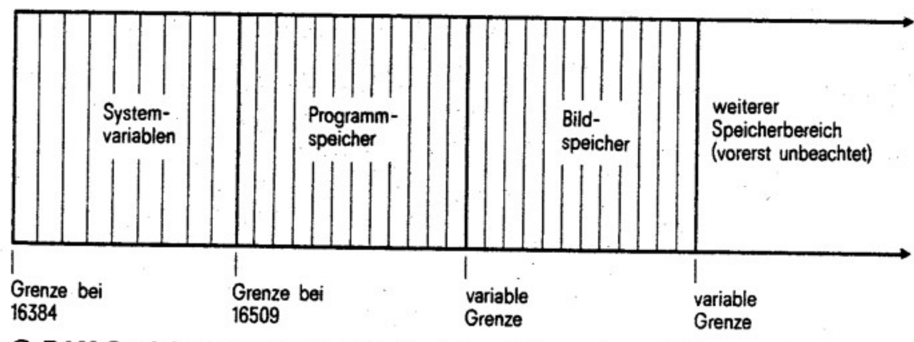
Versuchen Sie jetzt einmal selbständig, allein durch „poken" die REM-Zeile so zu ändern, daß auf dem Bildschirm
10 REM GHI PRINT Y
steht. Dabei ist zu beachten, daß PRINT im Bildspeicher Platz für mehrere Bytes beansprucht (fünf Buchstaben), der Befehl im Programmspeicher jedoch nur den Umfang von einem Byte hat. PRINT samt vor- und nachgestelltem Leerzeichen läßt sich deshalb mit dem Schlüsselwort-Code 245 in die REM-Zeile bringen.

Wer die Aufgabe gelöst hat, ist fürs erste fertig. Wer nicht, sollte alles nochmal durchackern.

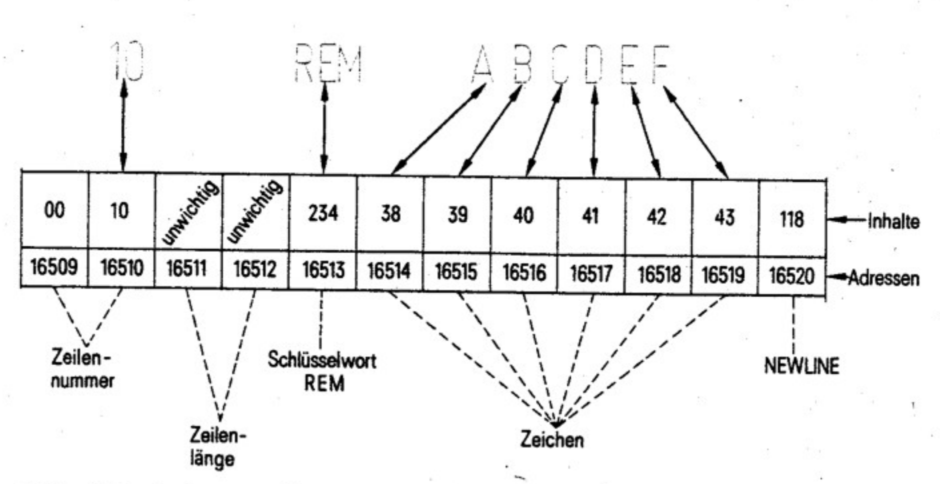
Im nächsten Teil geht es mit Systemvariablen weiter. Zur Vorinformation ist dafür eine kleine „Hausaufgabe" ratsam: Blättern Sie Ihr Anleitungsbuch zum ZX 81 durch und lesen Sie das Kapitel mit den Systemvariablen und das mit dem hexadezimalen Zahlensystem. Außerdem ist bald die Anschaffung eines Zusatzspeichers erforderlich, um die im Verlauf des Lehrgangs gegebenen Übungen nachvollziehen zu können: 4 KByte freier Speicherplatz sind dabei das Mindeste. Die FUNKSCHAU wird voraussichtlich in den Heften 12/13 eine Bauanleitung für einen passenden Zusatzspeicher veröffentlichen.

Klaus Herklotz

*(Wird fortgesetzt)*




② **RAM-Speicher des ZX 81:** Der Speicher ist in mehrere Sektoren eingeteilt




③ **Speicherbelegung:** So steht die Programmzeile „10 REM ABCDEF" im Programmspeicher des ZX 81

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 2: Bedeutung der Systemvariablen

**Nicht nur ein Programmierer kann Variablen Werte zuweisen – auch der Computer selbst kann das, wobei für ihn sogenannte Systemvariablen reserviert sind.**

Im Arbeitsspeicher des ZX 81 befindet sich der Bildspeicherbereich direkt hinter dem Programmspeicherbereich. Wenn nun der Bereich des Programmspeichers durch Eingabe von Programmzeilen anwächst, dann verschiebt sich der Bildspeicherbereich gezwungenermaßen. Im Gegensatz zu der festen Adresse, die den Beginn des Programmspeichers markiert, ist die Grenze zwischen diesen beiden Speicherbereichen variabel.

Benutzer des Computers können jedoch den momentanen Stand der Grenze erfahren, weil zwei 8-Bit-Speicherzellen mit dieser Information belegt werden. Die beiden Speicherzellen befinden sich im Bereich der Systemvariablen und tragen den Namen D-FILE.

### D-File beherbergt die Adresse des Bildspeicherbeginns

Die Systemvariable D-FILE belegt die Adressen 16396 und 16397 im Arbeitsspeicher. Der erste denkbare Schritt wäre, die Adressen zu lesen:
PRINT PEEK 16396, PEEK 16397
schreibt den Inhalt der beiden 8-Bit-Speicherzellen nebeneinander auf den Bildschirm. Wenn der Programmspeicher noch leer ist, sind das die Zahlen 125 und 64. Es stellt sich jetzt die Frage, was die beiden Zahlen bedeuten.

Beide Zahlen ergeben – richtig miteinander verknüpft – die Adresse, bei der der Bildspeicherbereich beginnt. Mikroprozessor-Architekten haben vereinbart, daß das hintere Byte das „meist signifikante“ Byte ist (Most Significant Byte, MSB). Man muß deshalb das Byte der höherwertigen Adresse (hier: 16397) mit 256 multiplizieren und zum Byte der niedrigeren Adresse (16396) addieren. Es ergibt sich 256 × 64 + 125 = 16509.

Zum Verständnis ein Beispiel: Jemand will seinem Gegenüber eine zweistellige Zahl mitteilen, kann aber immer nur eine Ziffer signalisieren. Beide haben deshalb vereinbart, daß der Empfänger der Zahlen immer zwei Ziffern hintereinander signalisiert bekommt, letztere mit 10 multipliziert und zur ersten addiert. Wird also die Ziffer 2 gefolgt von 6 signalisiert, so weiß der Empfänger, daß es sich um die Zahl 62 handelt.

Auf die gleiche Art geschieht dies beim ZX 81. Eine Adresse kann nicht in einer 8-Bit-Speicherzelle gespeichert werden, denn dazu ist sie mit ihren 16 Bit viel zu groß. Sie wird deshalb auf zwei 8-Bit-Speicherzellen aufgeteilt. Um die tatsächliche Adresse zu „konstruieren“, muß der Inhalt der hinteren Speicherzelle mit 256 multipliziert werden. Es ergibt sich deshalb aus den Zahlen 125 und 64 die Adresse 16509. Im Teil 1 war zu lesen, daß diese Adresse auch der unverrückbare Anfang des Programmspeichers ist. Das ist jedoch nicht verwunderlich, denn der Programmspeicher ist noch leer; somit beginnt dort der Bildspeicher.

Die durch Abfragen der Systemvariablen D-FILE gewonnene Adresse besagt also, daß die nächste Adresse (hier: 16510) die Adresse der Bildspeicherzelle links oben in der Ecke des Bildschirms ist. Die Eingabe
POKE 16510, CODE "A"
läßt dort ein A auftauchen. Nun ist aber zu bedenken, daß der ZX 81 in der Grundversion mit 1 KByte Speicherumfang lediglich 1024 Speicherzellen zur Verfügung stellt. Der ZX 81 ist deshalb in bezug auf den Bildspeicher zur Sparsamkeit erzogen worden: Wenn der RAM-Speicherplatz unter 3¼ KByte liegt (siehe Sinclair-Handbuch Kapitel 27), besteht bei leerem Bildschirm der Bildspeicher nur aus den 33 NEW-LINE-Codes vom Zeilenende (*Bild 1*).

Ohne Zusatzspeicher kann man zwar mit dem PRINT-Befehl arbeiten (der Bildspeicher wird dann um die erforderlichen Speicherbytes erweitert!), nichts bringt den Computer aber dazu, durch POKE 16510, CODE „A“ Platz für das A zu schaffen. Das A wird zwar noch geschrieben, doch wird dabei einer der NEW-LINE-Codes gelöscht, was den ZX 81 unweigerlich funktionsunfähig macht. Mit Zusatzspeicher (Bauanleitung voraussichtlich im nächsten Heft) erscheint das A ohne Wenn und Aber am Bildschirm.

**Vorsicht beim Poken!**

So reizvoll das Poken auch ist, so riskant ist es auch: Wird dem ZX 81 nämlich ein Byte an einer Speicherstelle aufgezwängt, deren ursprünglicher Inhalt für den internen Betriebsablauf des Computers maßgebend ist, dann protestiert der ZX 81, indem er verrückt spielt. Kuriose Bilder am Sichtschirm oder ein Löschen des Bildspeichers gepaart mit strikter Befehlsverweigerung sind das Ergebnis. Dem Computer schadet das nicht, aber seine Dienste können nur nach kurzfristigem Unterbrechen der Stromversorgung wieder in Anspruch genommen werden; dabei wird selbstverständlich auch das RAM gelöscht: Werden daher POKE-Befehle in ein längeres Basic-Programm eingebaut, dann sollte sicherheitshalber vor dem Starten mit RUN das Programm auf Magnetband gespeichert werden!

Geben Sie ein: POKE 16542,38
Nach dem zweiten NEW LINE zeigt der ZX 81 die kalte Schulter.

Wie Bild 1 zeigt, befinden sich dann in jeder Zeile des Bildschirms 33 Bildspeicherzellen. (Achtung: Zählung beginnt in Bild 1 bei 0.) Sie sind fortlaufend numeriert; letztes Zeichen in jeder Zeile ist NEW LINE. Das Fernsehbild nimmt also mit Zusatzspeicher bei 24 Zeilen 33 × 24 = 792 Byte Speicherplatz in Anspruch.

Um z. B. in der rechten unteren (auch durch PRINT AT noch erreichbaren) Ecke ein A zu schreiben, ist die Eingabe von

```
POKE 725 + PEEK 16396 + 256 *
PEEK 16397,38
```

erforderlich (725 = 33 × 22 − 1; A ≙ Code 38). Damit wird klar, daß nicht nur mit PRINT AT Punkte des Bildschirms erreichbar sind, sondern auch durch ein direktes Belegen der Bildspeicherzellen. Dies ist später für die Maschinensprache von Bedeutung.

## Aufspüren der aktuellen PRINT-Position

Die nächste wichtige Systemvariable heißt DF-CC. Sie gibt die Adresse der (letzten) PRINT-Position im Bildspeicher an und wird in den Adressen 16398 und 16399 abgelegt.

Mit dem folgenden Programm läßt sich feststellen, welche Adresse eine Bildspeicherzelle hat, von der man die Zeile und Spalte kennt:

```
10 PRINT AT 0,0:
20 LET A = PEEK 16398 + 256 *
   PEEK 16399
30 PRINT AT 10,10: A
```

Mit Zeile 30 wird die Adresse (A) der Bildspeicherzelle für die PRINT-Position nullte Spalte und nullte Zeile ausgegeben; der Wert ist nicht 16510, da jetzt der Programmspeicher nicht mehr leer ist. Die Eingabe

```
10 PRINT AT 3,4;
```

verändert die PRINT-Position im Bildspeicher: Jetzt wird die Adresse der Zelle in der vierten Spalte der dritten Zeile ausgegeben. Durch

```
POKE A,38
```

wird an dieser Position ein A geschrieben.

Die praktische Bedeutung dieser Systemvariablen liegt aber nicht im Poken, sondern im Peeken: Es läßt sich nämlich der Inhalt sämtlicher Bildspeicherzellen ohne nennenswerten Aufwand lesen.

Das folgende Listing verdeutlicht das:

```
10 PRINT AT 3,4; "ABCDEF"
20 PRINT AT 3,4:
30 LET A = PEEK 16398 + 256 *
   PEEK 16399
40 PRINT AT 20,0; PEEK A
```

Von dem Programm wird eine Buchstabenfolge ausgedruckt (Zeile 10). Danach wird die alte PRINT-Position wieder eingenommen (Zeile 20) und deren Adresse festgestellt (Zeile 30). Diese Adresse wird gelesen und der Inhalt – hier 38 ≙ A – ausgedruckt (Zeile 40). Um andere Speicherzellen zu lesen, muß z. B. nur die zweite Koordinate in Zeile 20 verändert werden.

## Der nutzbare Bildschirm wird größer

Die bisherigen Systemvariablen benötigen zum Unterbringen ihres Wertes zwei Adressen. Es gibt aber auch Systemvariablen, die sich mit einer Adresse begnügen: Sie werden deshalb Ein-Byte-Systemvariablen genannt.

Ein Beispiel dafür ist die Variable mit dem Namen DF-SZ. Sie hat die Adresse 16418 und ist ebenfalls für den Bildschirm zuständig. In DF-SZ ist die Anzahl an der Unterkante des Bildschirms normalerweise nicht verwendbarer Zeilen enthalten (belegbar sind nur 22 der 24 Zeilen).

Der Inhalt dieser Systemvariablen ist üblicherweise „2", das heißt, es gibt zwei leere Zeilen am unteren Bildschirmrand. DF-SZ bietet jetzt die Möglichkeit, den nutzbaren Bildschirmbereich um diese zwei Zeilen zu vergrößern:

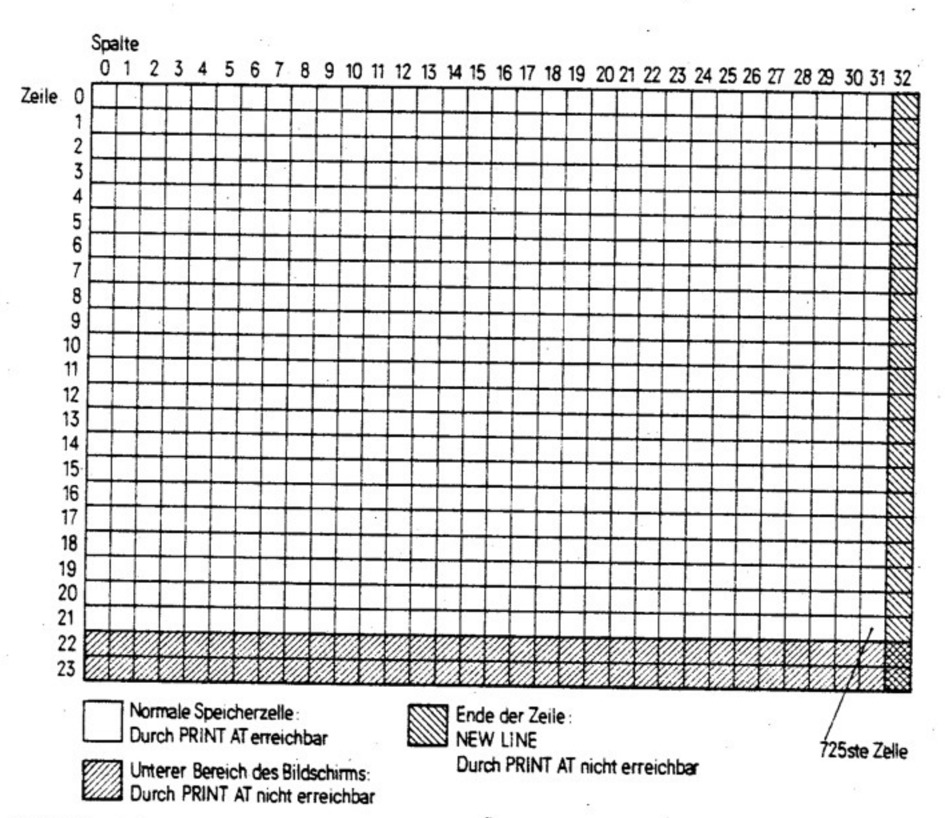
```
10 POKE 16418,0
20 PRINT AT 23,10; "ABCDEF"
```

Auf die gleiche Weise läßt sich der nutzbare Bildschirmbereich verkleinern, was aber nicht sinnvoll ist.

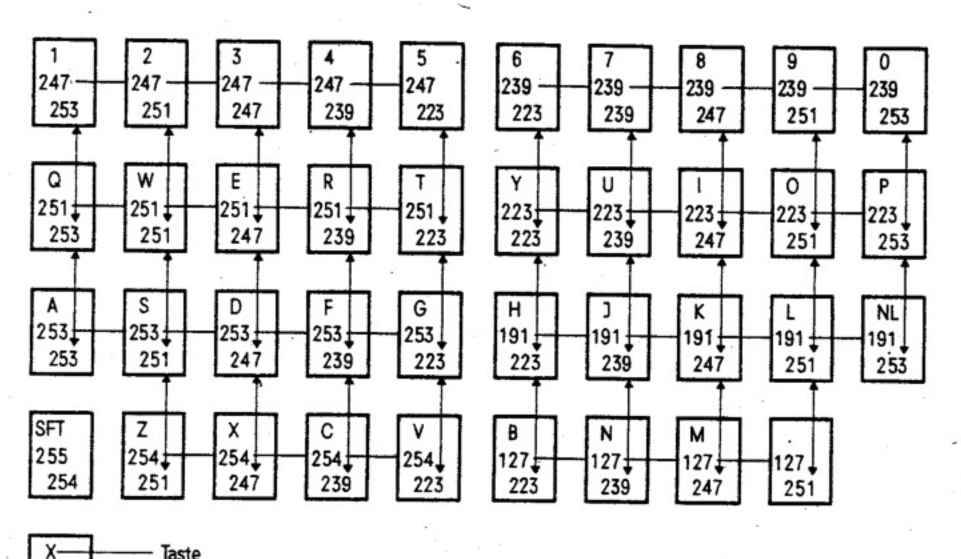
Im Grunde ist diese Systemvariable unwichtig für spätere Maschinenprogramme. Sie soll nur stellvertretend für die Systemvariablen gezeigt werden, mit denen der Benutzer des Computers das System selbst verändern kann.

## Welche Taste wurde zuletzt gedrückt?

Bis jetzt haben wir uns mit den Systemvariablen zur Datenausgabe beschäftigt. Es folgt nun eine zur Dateneingabe über das Tastenfeld: Ihr Name ist LAST-K. Sie ist in den Adressen 16421 und 16422 enthalten und zeigt an, welche Taste gerade gedrückt wird:




① **Bildschirmorganisation:** Jede der 704 normalen PRINT-Positionen kann ein Zeichen aus dem Zeichenvorrat des ZX 81 aufnehmen




② **Wertetabelle für LAST-K:** Abhängig davon, welche Taste *allein* gedrückt wird, ändern sich die Bytes unter den Adressen 16421 und 16422 (Systemvariable LAST-K) in der gezeigten Weise. Andere Werte ergeben sich, wenn zwei Tasten gleichzeitig gedrückt werden (z. B. SHIFT-Ebene)

```
10 PRINT AT 0,0; PEEK 16421,
   PEEK 16422
20 GOTO 10
```

Wenn eine Taste gedrückt wird, ändern sich die beiden ausgegebenen Zahlen. In *Bild 2* sind die Werte tabellarisch aufgelistet.

Vor allem ist diese Systemvariable wichtig für spätere Maschinenprogramme, denn mit ihr läßt sich jede gedrückte Taste lokalisieren. Hier eine Anwendung:

Es soll eine Programmzeile entworfen werden, die, in ein Programm eingefügt, dieses solange unterbricht, bis die Taste P gedrückt wird. Im Basic liegt die Lösung auf der Hand:

```
10 IF INKEY$ <> "P" THEN GOTO 10
```

Wenn man die Systemvariable LAST-K einsetzt, sieht die Lösung so aus:

```
10 IF PEEK 16421 <> 223 OR PEEK
   16422 <> 253 THEN GOTO 10
```

Wenn die Systemvariable LAST-K richtig eingesetzt wird, könnte auf die Funktion INKEY$ durchaus verzichtet werden!

## Systemvariable zur Zeitmessung

Außer den bisher vorgestellten Systemvariablen gibt es noch gut 30 andere, die für Maschinensprache größtenteils unwichtig sind.

Eine noch interessante Systemvariable trägt den Namen FRAMES und ist in den Speicherzellen 16436 und 16437 zu finden. FRAMES kann für Zeitmessungen verwendet werden. Beim Einschalten des ZX 81 nehmen die Inhalte beider Zellen den Wert 255 an. Danach wird 50mal je Sekunde das weniger signifikante Byte (also das Byte aus 16436) um Eins verringert. Ist es schließlich bei Null angelangt, so wird es wieder zu 255 und gleichzeitig wird das mehr signifikante Byte um Eins verringert. Wenn beide Bytes Null geworden sind, nehmen Sie wieder den Wert 255 an.

Das Ganze läßt sich so demonstrieren:

```
10 PRINT AT 0,0; PEEK 16436 + 256 *
   PEEK 16437
20 GOTO 10
```

Zusammenfassend läßt sich über die Systemvariablen sagen, daß sie die Möglichkeit geben, etwas über den momentanen Zustand des Systems selbst zu erfahren, manche sogar die Möglichkeit geben, das System zu verändern. In Maschinenprogrammen helfen sie beim Schreiben am Bildschirm und bei der Dateneingabe. Klaus Herklotz

(*Wird fortgesetzt*)

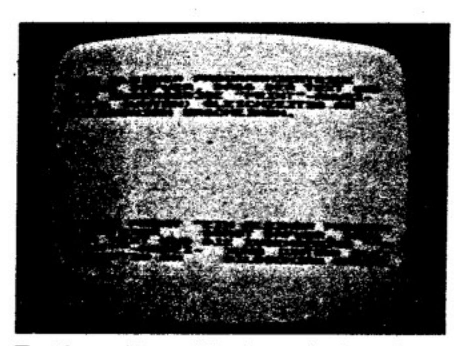
### ZX-81-Softwaretip:

# Textkorrektur – leicht gemacht

Wenn ein Programm den Computer veranlaßt, mit PRINT-Anweisungen längere Texte am Bildschirm auszugeben, dann ist das Ergebnis auf dem Bildschirm meist alles andere als schön: Trennungsstriche fehlen, die Worttrennung erfolgt an der falschen Stelle und alle Zeilen beginnen am linken Bildschirmrand, egal wie lang sie sind. Von einem gegliederten Textaufbau kann also keine Rede sein. Da der Zeilenfall bei der PRINT-Anweisung völlig anders ist als der am Bildschirm wiedergegebene, gelingt das richtige Einfügen von Trennungsstrichen und Leerzeichen oft erst nach mehreren Anläufen.

Schneller geht es, wenn die Programmzeile mit der PRINT-Anweisung und die ausgeführte PRINT-Anweisung *gleichzeitig* am Bildschirm erscheinen, da sich die Auswirkung einer Textänderung dann sofort am Zeilenfall nachprüfen läßt (*Bild*).

Das ist kein Problem, wenn nach der PRINT-Anweisung in der nächsten Programmzeile ein STOP-Befehl vorübergehend eingefügt wird. Dann bewegt man im Programmlisting den Cursor nach oben bis zur Zeilennummer der PRINT-Anweisung. Gestartet mit RUN wird nun das Programm bis zum STOP-Befehl ausgeführt, der Text der PRINT-Anweisung auf den Bildschirm geschrieben und die Meldung 9/xxx ausgegeben. Jetzt ist nur noch EDIT einzugeben, um auch die Programmzeile mit der PRINT-Anweisung auf den Bildschirm zu holen. Nun kann nach Herzenslust korrigiert werden. -ll



**Textkorrektur:** Mit einem kleinen Trick läßt sich gleichzeitig zur Textanzeige die entsprechende PRINT-Anweisung auf den Bildschirm holen

# Klartext für den ZX 81

## Teil 3: „Hex" ist keine Hexerei

**Keine Angst, hier geht es nicht zum x-ten Male um Sinn und Zweck des hexadezimalen Zahlensystems, sondern um den praktischen Umgang mit Hex-Codes.**

Wer sich von der Pike aufwärts mit dem hexadezimalen Zahlensystem beschäftigen will, sei auf eines der vielen Mikrocomputer-Grundlagenbücher verwiesen. Wir hier setzen Grundkenntnisse voraus, zumal sogar das ZX-Handbuch ein eigenes Kapitel diesem Problem widmet. Der sichere Umgang mit dem hexadezimalen Zahlensystem ist und bleibt freilich beim Programmieren in Maschinensprache von allergrößter Wichtigkeit, so daß nachfolgend verschüttetes Wissen anhand praktischer Beispiele aufpoliert wird.

Als Einstieg sei kurz angemerkt, daß zur Darstellung eines Bytes im dezimalen Zahlensystem drei Stellen notwendig sind. Diese drei Stellen werden aber recht unbefriedigend genutzt: Es werden nur die Zahlen 0 bis 255 benötigt, obwohl 999 als größte dreistellige Zahl möglich wäre. Im hexadezimalen Zahlensystem braucht man zum Darstellen eines Bytes nur zwei Stellen, die jedoch restlos genutzt werden: FF ist damit die größte Zahl, die zur Darstellung eines Bytes notwendig ist.

### Umwandeln per Programm

Weil die hexadezimalen Zahlen große Bedeutung haben, aber der Computer beim Poken nur Dezimalzahlen annimmt (bei der Adresse und beim Byte), stellt sich das Problem der Umwandlung von einem Zahlensystem ins andere. Bei Ein-Byte-Zahlen ist dies einfach. Zur Not reicht dazu der Anhang A des ZX-Handbuchs.

Bei größeren Zahlen beginnen aber die Schwierigkeiten: Wie lautet z. B. die Zahl 16514 in hexadezimaler Schreibweise? Das ist wirklich eine harte Nuß. Ein Umwandlungsprogramm (*Bild 1*) erleichtert die Arbeit ungemein. Es ist daher empfehlenswert, das Programm auf Kassette aufzunehmen.

Zum Umgang mit dem Programm: durch Drücken der NEW-LINE-Taste wird es abgebrochen. Außerdem nimmt der Rechner nur fünfstellige Dezimalzahlen und vierstellige Hexadezimalzahlen an: Kleinere Zahlen müssen durch Anfangsnullen auf die richtige Stellenzahl gebracht werden.

Für einen ersten Test nehmen wir die Zahl 16514: Wenn das Programm fehlerfrei eingegeben wurde, muß „4082H" ausgegeben werden. Gemeint mit Dezimalzahlen zu vermeiden).

In Zeile 110 wird der eingegebene String I$ (String: Zeichenfolge) auf die Länge hin überprüft: Nur Strings der Länge 5 werden zugelassen und auf dem Bildschirm ausgegeben. In Zeile 120 wird dann ermittelt, ob die letzte Stelle des Strings ein H ist. Bei 16514 ist dies nicht der Fall: Weiter geht's deshalb ohne Sprung. Die Variable I erhält nun den Wert von I$.

Mit Zeile 140 werden die hinteren beiden Stellen (der zukünftigen Hexadezimalzahl) ermittelt und von Zeile 280 in eine Hexadezimalzahl umgewandelt. Dieser Wert wird unter I$ gespeichert und anschließend werden die vorderen beiden Stellen umgewandelt (Zeilen 170 und 180). In Zeile 190 wird der Ergebnisstring zusammengesetzt und mit Zeile 200 ausgegeben.

Als nächstes der umgekehrte Fall: 4082H wird in 16514 umgewandelt. In Zeile 120 wird diesmal festgestellt, daß eine Hexadezimalzahl eingegeben wurde: es erfolgt ein Sprung nach 300. Dort ermittelt der Rechner wieder die ersten beiden Stellen und wandelt sie in Zeile 480 in eine Dezimalzahl um. Diese Zahl wird in Zeile 320 unter der Variablen I gespeichert. Danach werden die hinteren beiden Stellen ermittelt und umgewandelt. In Zeile 350 ermittelt der Rechner schließlich die endgültige Zahl I.

Sehr wichtig, vor allem zum Verständnis des nächsten Programms, ist die Zeile 480: Dort wird ein String der

```
100 INPUT I$
105 IF I$="" THEN STOP
110 IF LEN I$<>5 THEN RUN
115 PRINT I$;" = ";
120 IF I$(5)="H" THEN GOTO 300
130 LET I=VAL I$
140 LET Z=I-256*INT (I/256)
150 GOSUB 280
160 LET I$=Z$
170 LET Z=INT (I/256)
180 GOSUB 280
190 LET I$=Z$+I$+"H"
200 PRINT I$
210 RUN
280 LET Z$=CHR$ (28+INT (Z/16))+CHR$ (Z-16*INT (Z/16)+28)
290 RETURN
300 LET Z$=I$(1 TO 2)
310 GOSUB 480
320 LET I=Z
330 LET Z$=I$(3 TO 4)
340 GOSUB 480
350 LET I=256*I+Z
360 PRINT I
370 RUN
480 LET Z=16*(CODE Z$(1)-28)+CODE Z$(2)-28
490 RETURN
```

① **Umwandlungsprogramm:** Damit werden Hexadezimalbzw. Dezimalzahlen fürs jeweils andere Zahlensystem umgewandelt

Länge Zwei (mit einer Ein-Byte-Hexadezimalzahl) in eine Dezimalzahl umgewandelt. Gehen wir davon aus, daß es sich dabei um die Zahl FAh handelt.

Die Zahl FAh hat zwei Stellen, eine mehr signifikante (das F) und eine weniger signifikante (das A). Zuerst wird in der Zeile 480 der Code der mehr signifikanten Stelle ermittelt; er ist 43. Davon wird 28 abgezogen und man erhält die entsprechende Dezimalzahl für F, nämlich 15. Da es sich um die mehr signifikante Stelle handelt, muß man das Ergebnis mit 16 multiplizieren.

Durch Umwandlung des A erhält man dessen dezimalen Wert 10, der zum vorhergehenden Ergebnis addiert werden muß. Das endgültige Ergebnis ist die Zahl 250.

## Zurück zum Poken einer REM-Zeile

Die Methode eine REM-Zeile zu Poken (siehe Teil 1) hat einen entscheidenden Vorteil gegenüber der direkten Eingabe der REM-Zeile: Sogar die Fragezeichen-Symbole der nicht benutzten Codes können in der REM-Zeile erscheinen. Wie sonst könnte eine Speicherzelle den Inhalt 96 erhalten, wenn nicht durch Poken?

Das Poken soll jetzt vereinfacht werden. Ausgangspunkt ist die Zeile
10 REM 00000000
Diese Zeile soll zu
10 REM BDAFGHIK
verändert werden. Natürlich kann wieder jede Speicherzelle einzeln gepoked werden. Leichter ist es aber, eine Schleife zu programmieren und die Arbeit dem Computer zu überlassen.

*Bild 2* zeigt das Programmlisting: In A$ sind die hexadezimalen Codes der Zeichen BDAFGHIK in der richtigen Reihenfolge enthalten. Danach wird die Anfangsadresse auf 16514 gelegt, das ist die Adresse des ersten Zeichens in der REM-Zeile. In der Schleife ab Zeile 40 werden die einzelnen Hexadezimal-Codes aus A$ umgewandelt und eingeschrieben.

Der hintere Teil der Zeile 50 hat große Ähnlichkeit mit der Zeile 480 des letzten Programms, er hat auch die selbe Funktion. Soll eine andere Zeichenfolge in der REM-Zeile erscheinen, so müssen nur die Codes im String verändert werden.

Spätestens jetzt stellt sich die Frage, was die REM-Zeile mit dem Maschinencode, um den es letztendlich geht, zu tun hat.

## Maschinenprogramme haben einen eigenen Startbefehl

Wie das Wort „Maschinencode" schon sagt, handelt es sich bei dieser Art des Programmierens um ein Programmieren mit Codes. Das heißt, daß jeder Befehl in Maschinensprache nicht durch ein Schlüsselwort (Keyword) wie in Basic, sondern durch eine Hexadezimalzahl von der Größe eines Bytes dargestellt wird.

In einer REM-Zeile sind Zeichen enthalten, deren Codes ebenfalls den Umfang eines Bytes haben. Es ist daher vorstellbar, ein ganzes Maschinenprogramm in eine REM-Zeile zu schreiben. Hier kann man schon den ersten Vergleich von Maschinensprache mit Basic anstellen: In Basic erfolgt die Eingabe eines Programmes durch Eintippen von Programmzeilen. In Maschinensprache dagegen wird ein Programm gerne durch Poken einer REM-Zeile eingegeben. Weiterhin wird ein Basic-Programm durch den Befehl RUN zum Laufen gebracht. Wie aber bringt man ein Maschinenprogramm (was das genau ist und wie es aussieht ist noch unbedeutend!) zum Laufen?

Aus früheren Überlegungen heraus steht fest, daß das erste Zeichen nach dem REM-Befehl in Adresse 16514 steht, wenn die REM-Zeile die erste Programmzeile ist.

Das Maschinenprogramm beginnt also bei Adresse 16514. Die Möglichkeit dort ein Maschinenprogramm aufzurufen bietet die USR-Funktion:

LET Q=USR 16514

befiehlt dem Rechner, ein in Maschinensprache geschriebenes Programm ab Adresse 16514 auszuführen. Die Wirkung ist mit GOSUB16514 in Basic zu vergleichen.

Jetzt fehlt nur noch das Maschinenprogramm selbst. In *Bild 3* ist ein Basic-Programm aufgelistet, das die Eingabe von Maschinenprogrammen und deren Aufruf erleichtert. Es enthält bereits eine lange REM-Zeile für lange Maschinenprogramme und sollte deshalb durch GOTO 80 auf Kassette gespeichert werden.

Das erste Maschinenprogramm, das vorgestellt wird, soll nichts tun! Im Z-80-Maschinencode gibt es tatsächlich den Befehl „nichts tun" (englisch: *no operation*). Er hat die Abkürzung „nop" und den Hexadezimal-Code 00. Im Programm (Bild 3) muß also A$="" zu A$="00" verändert werden. Durch RUN wird das „Programm" in die REM-Zeile übernommen und durch LET Q=USR 16514 zum Laufen gebracht.

Hier gibts gleich die erste Überraschung: Der Computer steigt aus! Warum das so ist und wie man es vermeiden kann, verrät der nächste Teil.

Klaus Herklotz
*(Wird fortgesetzt)*

```
10 REM 00000000
20 LET A$="2729262B2C2D2E30"
30 LET ADR=16514
40 FOR N=0 TO LEN A$-2 STEP 2
50 POKE ADR+INT (N/2),16*(CODE A$(N+1)-28)+CODE A$(N+2)-28
60 NEXT N
```

② **Poken einer REM-Zeile:** Dieses Hilfsprogramm erspart das mühsame Poken einzelner Adressen

```
10 REM 000000000000000000000000000000000000000000000000000000
20 LET A$=""
30 LET ADR=16514
40 FOR N=0 TO LEN A$-2 STEP 2
50 POKE ADR+INT (N/2),16*(CODE A$(N+1)-28)+CODE A$(N+2)-28
60 NEXT N
70 STOP
80 SAVE "USR"
90 LIST
95 STOP
```

③ **Eingabe von Maschinenprogrammen:** Mit diesem Programm lassen sich beliebige Zeichen des ZX 81 in der REM-Zeile unterbringen. Nach dem Laden wird es automatisch aufgelistet

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 4: Datentransport mit „load"

**In Basic weist der Befehl LET einer Variablen einen Wert zu. In Maschinensprache übernimmt der Befehl load diese Aufgabe.**

Die letzte Folge endete mit einem Mißerfolg: Schon das einfachste Maschinenprogramm brachte den ZX 81 außer Kontrolle. Dies soll eine Warnung sein. Es darf nie ein Maschinenprogramm ohne Rücksprung abgeschlossen werden.

Durch LET Q = USR 16514 wurde unser Maschinenprogramm bei Adresse 16514 aufgerufen. Der Computer verläßt damit den Basic-Modus und interpretiert jedes Zeichen ab dieser Adresse als Maschinensprachebefehl. Erfolgt keine Rückkehr in den Basic-Modus, so weiß der Computer nicht mehr weiter und steigt aus oder liefert falsche Ergebnisse!

### Zurück ins Basic

Im Z-80-Maschinencode gibt es für den Rücksprung ins Basic den Befehl „return", was in Basic dem Rücksprung aus einem Unterprogramm gleichkommt. Das Befehlskürzel ist „ret" und der Hexadezimalcode C9.

Verwendet wird dieser Befehl vorerst als Abschluß eines Maschinenprogramms. Im Eingabeprogramm der letzten Folge muß A$ = "00" also zu A$ = "00C9" verändert werden. Durch RUN wird das Programm in die REM-Zeile gebracht und dann durch LET Q = USR 16514 zum Laufen gebracht. Diesmal bleibt der Computer bei der Stange und erledigt den Auftrag des „nichtstuns" ohne weiteres. Was geht hierbei im Computer vor?

Durch USR 16514 springt er in die Maschinenroutine bei Adresse 16514. Den Inhalt dieser 8-Bit-Speicherzelle faßt der Computer als Befehl auf: 00 heißt „no operation". Das bedeutet, er kann sich der nächsten Speicherzelle zuwenden. Dort findet der Computer C9 und faßt es als return-Befehl auf: Rücksprung nach Basic.

Bei der Dokumentation längerer Maschinenprogramme wäre es sehr unübersichtlich, wenn einfach der String A$ = "...C9" mit hexadezimalen Bytes angegeben wird. Eine für den Anwender besser durchschaubare Form hat die allgemein übliche Notation von Maschinenprogrammen (*Bild 1*): Der hexadezimal angegebenen Adresse folgt ihr Inhalt (Byte). Der Übersicht halber können pro Zeile bis zu vier Bytes stehen. Warum das so ist wird später behandelt.

In der Spalte Z-80-Assembler steht schließlich die Übersetzung der Bytes in den sogenannten Assemblercode: nop heißt no operation und ret steht für return. Die Spalte Label ist noch unbedeutend.

### Im Maschinencode kennt der ZX 81 Pseudo-Variablen

Im ZX-81-Basic kann der LET-Befehl jeder beliebigen Variablen einen Wert von −10E38 bis 10E38 zuweisen. Zum Beispiel weist LET B = 120 der Variablen B den Wert 120 zu.

In der Z-80-Maschinensprache stehen dem Benutzer sieben solcher Variablen zur Verfügung. Sie werden aber nicht Variable sondern „Register" genannt (*Bild 2*). Ein weiterer Unterschied zu Basic: Jedes Register hat den Umfang eines Bytes, kann also nur Werte von 00 bis FFh aufweisen oder zugeordnet bekommen.

Ein Byte-Umfang ist zur Zahlenverarbeitung freilich etwas wenig. Es gibt daher die Möglichkeit, Register zu einem Registerpaar zusammenzufassen. Daraus entstehen dann das bc-Registerpaar, das de- und das hl-Registerpaar. Sie haben einen Umfang von zwei Bytes, können also Zahlen von 0000 bis FFFFh aufnehmen.

Auch der LET-Befehl hat in Maschinensprache einen anderen Namen; dort ist vom Datentransportbefehl load (laden) mit dem Kürzel „ld" die Rede.



| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | 00 | | | | | nop |
| 4083 | C9 | | | | | ret |

① **Listing zum Programm „Nichts tun":** In dieser Form lassen sich Maschinenprogramme gut dokumentieren



② **Sieben Register des Z 80:** Im Maschinencode lassen sich den Registern Zahlen unmittelbar zuordnen. Das a-Register (Akkumulator) spielt bei den meisten Operationen eine maßgebende Rolle

| ADRESSE | BYTES | | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|---|
| 4082 | 06 | 00 | | | | | ld b,00 |
| 4084 | 0E | 01 | | | | | ld c,01 |
| 4086 | C9 | | | | | | ret |

③ **Listing für das Laden des bc-Registerpaares:** Das Registerpaar wird mit der Zahl 0001 geladen

So ist z. B. „ld c, 01" gleichbedeutend mit LET C = 1: Das c-Register erhält den Wert 01 zugewiesen. Für ld c, 01 gibt es aber keinen eigenen Code, wie z. B. C9 für ret (return). Es gibt nur den Code für ld c, N. Das große N steht stellvertretend für eine beliebige Zahl zwischen 00 und FFh. ld c, N hat den Code 0Eh. Die Zahl N selber steht in der nächsten 8-Bit-Speicherzelle. Um dem c-Register den Wert 01 zuzuweisen muß also der String zu A$ = "...0E01..." erweitert werden.

## Das bc-Registerpaar läßt sich auslesen

Maschinenprogramme in einer REM-Zeile werden z. B. durch LET Q = USR 16514, RAND USR 16514 oder PRINT USR 16514 aufgerufen. Als Verknüpfungsglied zwischen der Basic- und der Maschinenebene erhält dann der Ausdruck USR 16514 nach dem Rücksprung durch ret aus dem Maschinenprogramm, das bc-Registerpaar zugewiesen. Im Klartext heißt das:

PRINT USR 16514 führt ein Maschinenprogramm ab Adresse 16514 durch und gibt danach den Inhalt des bc-Registerpaares am Bildschirm aus. Damit läßt sich der Inhalt dieses Registerpaares lesen.

## Das erste richtige Maschinenprogramm

Jetzt wird ein Maschinenprogramm entworfen, daß das bc-Registerpaar mit 0001 lädt. Dazu sind zwei Schritte notwendig: Als erstes ist das b-Register mit 00 zu laden und danach das c-Register mit 01.

Für „ld c, N" ist der Code 0Eh. Für „ld b, N" ist er 06h. Eine Zusammenstellung der Maschinenbefehle enthält der Anhang A (Zeichenvorrat) des Sinclair-ZX-Handbuchs.

Das vollständige Listing des Maschinenprogramms zum Lösen der Aufgabe zeigt *Bild 3*. Demnach muß der String des Eingabeprogrammes zu A$ = "06000E01C9" verändert werden. Nach dem „Einpoken" durch RUN und dem Aufruf durch PRINT USR 16514 wird der Wert des bc-Registerpaares, nämlich „1" ausgegeben. Verfolgen wir wieder den Programmablauf aus der Sicht des Mikroprozessors:

PRINT USR 16514 schickt den Computer in den Maschinensprachemodus zur Adresse 16514. Ihr Inhalt ist der erste Befehl: 06 gebietet dem Computer, das b-Register mit der Zahl zu laden, die unter der nächsten Adresse zu finden ist. Das ist 00, womit die erste Operation bereits beendet ist. Der Prozessor faßt den Inhalt der nächsten Adresse wieder als Befehl auf: 0E veranlaßt ihn, das c-Register mit dem Inhalt der folgenen Adresse zu laden. Dort steht 01 und beendet den zweiten Schritt. Den Rücksprung nach Basic und den Ausdruck des bc-Registerpaares gebietet letztendlich C9 sowie der Aufruf des Maschinenprogramms durch PRINT USR.

Was aber wird ausgedruckt, wenn das c-Register mit 00 und das b-Register mit 01 geladen wird? Die Antwort liefert der Rechner. Vor Eingabe eines neuen Codes in Zeile 20 sollte die REM-Zeile sicherheitshalber wieder mit Nullen gefüllt werden, damit nicht Reste eines alten (längeren) Programms erhalten bleiben.

Genau wie die Register b und c, können auch die anderen Register mit einer Zahl geladen werden. *Bild 4* zeigt die Befehle mit den Codes. Für jede dieser Operationen werden zwei Speicherzellen benötigt: Eine für den Befehl und eine für die Zahl. Es handelt sich daher um Zwei-Byte-Befehle.

## Kopierbefehle decken die übrigen Register auf

Mit den bisherigen Mitteln (Rücksprung mit Ausdruck des bc-Registerpaares) ist nicht feststellbar, welche Zahlen in den Registern d, e, h und l sowie im Register a (Akkumulator) stehen. Daher kommen als nächstes Befehle an die Reihe, mit denen einzelne Register in andere geladen werden, das heißt kopiert werden.

Der Befehl ld c, h lädt das h-Register ins c-Register. Dabei geht der Inhalt des h-Registers nicht verloren: Nach der Befehlsausführung ist das Byte des h-Registers sowohl im c-Register als auch im h-Register gespeichert! Der Befehl ld c, h wäre also mit LET C = H in Basic zu vergleichen. Eine Zusammenstellung der Kopier-Befehle zeigt *Bild 5*.

Mit Hilfe dieser Befehle können wir jetzt auch feststellen welchen Inhalt z. B. das d-Register hat. Es muß nur das d-Register ins c-Register kopiert werden (b-Register mit 00 laden), denn der Rücksprung nach Basic bringt bekanntlich den Inhalt des bc-Registerpaares an den Tag (ret nicht vergessen). Genaueres steht im nächsten Teil.

Klaus Herklotz

*(Wird fortgesetzt)*

| Befehl | Code |
|---|---|
| ld a, N | 3E N |
| ld b, N | 06 N |
| ld c, N | 0E N |
| ld d, N | 16 N |
| ld e, N | 1E N |
| ld h, N | 26 N |
| ld l, N | 2E N |

④ **Ladebefehle des Z 80:** Jedes der sieben Register hat zum Laden einer zweistelligen Hexzahl (N) einen eigenen Ladebefehl

| r = | a | b | c | d | e | h | l |
|---|---|---|---|---|---|---|---|
| ld a, r | 7F | 78 | 79 | 7A | 7B | 7C | 7D |
| ld b, r | 47 | 40 | 41 | 42 | 43 | 44 | 45 |
| ld c, r | 4F | 48 | 49 | 4A | 4B | 4C | 4D |
| ld d, r | 57 | 50 | 51 | 52 | 53 | 54 | 55 |
| ld e, r | 5F | 58 | 59 | 5A | 5B | 5C | 5D |
| ld h, r | 67 | 60 | 61 | 62 | 63 | 64 | 65 |
| ld l, r | 6F | 68 | 69 | 6A | 6B | 6C | 6D |

⑤ **Kopierbefehle des Z 80:** Jedes Register kann den Inhalt jedes anderen Registers übernehmen. Z. B. kopiert der Befehl 78 das b- ins a-Register

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 5: Ladekommandos für Register

**Weiter geht's mit dem Laden von Registern. Die CPU befolgt hierbei jede Menge von Ladekommandos.**

Wie man einzelne Register mit Zahlen lädt ist sicher noch vom vorherigen Teil bekannt. Es soll nun das hl-Registerpaar mit der Hexadezimalzahl 400E geladen werden. Ob diese Operation gelungen ist, läßt sich nicht so einfach nachprüfen. Dazu muß erst das hl-Registerpaar ins bc-Registerpaar kopiert werden und ein Rücksprung zum Basic erfolgen; schließlich wird immer nur der Inhalt des bc-Registerpaars dem Basic gemeldet!

Es empfiehlt sich, das Programm (*Bild 1*) über das in Teil 3 beschriebene Eingabeprogramm einzugeben und es durch PRINT USR 16514 abzurufen. Wenn alles glatt gegangen ist, wird 16398 (400Eh) ausgegeben.

Bis jetzt ist es uns nur möglich, ein Registerpaar mit *zwei* Schritten zu laden, wozu insgesamt vier Bytes notwendig sind. Schuld daran haben die 8-Bit-Ladebefehle. Wie *Bild 2* zeigt, ermöglichen 16-Bit-Ladebefehle das Laden eines Registerpaares in einem Schritt mit nur drei Bytes: Die beiden Speicherzellen nach der Speicherzelle des Befehlscodes enthalten dann die zu ladende Zahl. Dabei wird das Byte direkt hinter dem Befehlscode ins niedrigerwertige Register geladen (0E ins c-Register!) und das letzte Byte ins höherwertige Register (40 ins b-Register!). Nachfolgend die 16-Bit-Ladebefehle auf einen Blick:

```
ld bc, NN  01
ld de, NN  11
ld hl, NN  21
```

Schreiben Sie zur Übung das Programm aus Bild 1 mit einem 16-Bit-Ladebefehl.

### „Peeken" in Maschinensprache

Jetzt beherrschen wir bereits die direkte Adressierung beim Ladevorgang, wobei Register direkt mit Zahlen oder dem Inhalt anderer Register geladen werden. Speziell für den Akkumulator gibt es jedoch weitere Ladebefehle:

Basic bietet die Möglichkeit, den Inhalt beliebiger Speicherzellen zu erfahren. Mittel dafür ist die PEEK-Funktion:

LET A=PEEK 16513

Diese Operation weist der Variablen A den Inhalt der Speicherzelle 16513 zu. In ähnlicher Weise funktioniert das auch in der Z-80-Maschinensprache. Das a-Register (Akkumulator) soll z. B. den Inhalt der Speicherzelle 16513 erhalten. In der Assemblerschreibweise sieht das so aus:

ld a, (4081h)

Die Klammer um die Zahl 4081h (16513) weist darauf hin, daß der *Inhalt* der Adresse 4081h gemeint ist. Genau wie bei den 16-Bit-Ladebefehlen sind auch hier drei Bytes notwendig: 3Ah als Kürzel von ld a, (NN) sowie die beiden Adreß-Bytes (wieder in vertauschter Reihenfolge). Das Maschinenlisting zu dieser Aufgabe ist *Bild 3* zu entnehmen: Nach dem Rücksprung wird 234 ausgegeben. Und so ist es dazu gekommen: Nach dem Sprung in die Maschinenebene zur Adresse 4082h erhält der Computer die Anweisung, den Akkumulator mit dem Inhalt der Speicherzelle 4081h zu laden. In dieser Speicherzelle – das ist früheren Überlegungen zu entnehmen – befindet sich der Code des REM-Befehls, nämlich EAh (234). Danach wird das b-Register mit 00 und das c-Register mit dem Akkumulatorinhalt geladen, so daß vor dem Rücksprung die Zahl 00EAh (234) im bc-Registerpaar steht.

| ADRESSE | BYTES | | | | Z-80-ASSEMBLER |
|---|---|---|---|---|---|
| 4082 | 2E | 0E | | | ld l, 0E |
| 4084 | 26 | 40 | | | ld h, 40 |
| 4086 | 4D | | | | ld c, l |
| 4087 | 44 | | | | ld b, h |
| 4088 | C9 | | | | ret |

① **Laden des hl-Registers:** Ob 400Eh tatsächlich geladen wurde, zeigt sich nach dem Kopieren ins bc-Register

| 8-bit-Befehle | | 16-bit-Befehle |
|---|---|---|
| ld c, 0E<br>ld b, 4,0 | → | ld bc, 400E |
| 0E 0E 06 40 | Bytefolge | 01 0E 40 |

② **16-Bit-Ladebefehle:** Bei gleicher Wirkung wird gegenüber 8-Bit-Befehlen ein Byte eingespart

| ADRESSE | BYTES | | | | Z-80-ASSEMBLER |
|---|---|---|---|---|---|
| 4082 | 3A | 81 | 40 | | ld a, (4081) |
| 4085 | 06 | 00 | | | ld b, 00 |
| 4087 | 4F | | | | ld c, a |
| 4088 | C9 | | | | ret |

③ **Erweiterte Adressierung:** Jetzt wird der Akku mit dem Inhalt der Adresse 4081h geladen

## Durch die Blume adressieren

Im wesentlichen gleicht die indirekte Adressierung der erweiterten Adressierung. Nur wird diesesmal die Adresse nicht direkt durch eine 16-Bit-Zahl dargestellt, sondern – wie der Name schon sagt – indirekt durch ein Registerpaar. So bewirkt z. B. der Befehl ld b, (hl), daß das b-Register mit dem Inhalt der Adresse geladen wird, die im hl-Registerpaar steht. Nachfolgend sind der Befehlsvorrat zur indirekten Adressierung sowie die dazugehörigen Codes aufgelistet:

| Befehl | Code | Befehl | Code |
|---|---|---|---|
| ld a, (NN) | 3A | ld c, (hl) | 4E |
| ld a, (bc) | 0A | ld d, (hl) | 56 |
| ld a, (de) | 1A | ld e, (hl) | 5E |
| ld a, (hl) | 7E | ld h, (hl) | 66 |
| ld b, (hl) | 46 | ld l, (hl) | 6E |

## Dem „Spielball" auf die Sprünge helfen

Mit den bisher gewonnenen Kenntnissen in Maschinensprache ist es möglich, einfache Probleme zu bewältigen: Nehmen wir an, es soll ein Programm entworfen werden, bei dem sich ein Spielball auf dem Bildschirm hin- und herbewegt. Wenn der Ball auf ein Hindernis trifft, soll er seine Richtung ändern. In Basic gibt es dann die Möglichkeit, die Koordinaten des Balles und die der Hindernisse zu vergleichen und bei Gleichheit eine Richtungsänderung einzuleiten. Dies nimmt jedoch, sobald die Zahl der Hindernisse größer wird, enorm viel Zeit und Speicherplatz in Anspruch. Einfacher ist es nachzuschauen, ob sich auf dem Feld, auf das der Ball als nächstes gelangt, ein Hindernis befindet. Wenn ja, dann soll der Ball die Richtung wechseln.

```
200 REM ANFANGSKOORDINATEN:
210 LET H=8
220 LET B=4
230 REM BEWEGUNGSRICHTUNG:
240 LET FH=1
250 LET FB=1
260 REM HINDERNISSE:
270 PRINT AT 2,4;"H";AT 7,1;"B";
    AT 17,13;"H";AT 12,16;"B"
300 REM ***HAUPTSCHLEIFE***
310 REM NAECHSTE PRINT-POSITION:
320 PRINT AT H+FH,B+FB;
330 REM FELDUNTERSUCHUNG:
340 LET Q=PEEK (PEEK 16398+
    256*PEEK 16399)
350 REM RICHTUNGSAENDERUNG:
360 IF Q=CODE "H" THEN LET FH=-FH
370 IF Q=CODE "B" THEN LET FB=-FB
380 REM DRUCKEN + LOESCHEN:
390 PRINT AT H,B;" "
410 PRINT AT H+FH,B+FB;"O"
420 REM KOORDINATNEN FIXIEREN:
430 LET H=H+FH
440 LET B=B+FB
450 GOTO 300
```

④ **Listing „Spielball":** Ein Ball bewegt sich damit zwischen vier Pfosten noch recht gemächlich

Zuerst die Lösung in Basic (*Bild 4*): Bis Zeile 270 werden die Variablen definiert und das „Spielfeld" gedruckt. Zeile 320 bringt die PRINT-Position auf das *voraussichtlich* nächste Feld. Dieses Feld wird in Zeile 340 in altbekannter Manier (siehe Teil 2) durch Abfragen einer Systemvariablen untersucht und dann eventuell die Bewegungsrichtung geändert. Der alte Spielball wird gelöscht (Zeile 390), der neue in Zeile 410 gedruckt und letztendlich die Koordinaten für den nächsten Durchgang festgelegt.

Damit sich das Lernen der Adressierungsarten auch gelohnt hat, versuchen wir jetzt Zeile 340 durch eine Maschinenroutine zu ersetzen. Das Eingabeprogramm läßt sich jetzt natürlich nicht verwenden (hinzuladen), weil sonst das Basic-Programm gelöscht wird. *Bild 5* enthält deshalb außer dem Flußdiagramm auch noch ein komplettes Eingabeprogramm für den Maschinenteil. *Bild 6* zeigt das Maschinenlisting.

Ab Adresse 4082h wird der Akkumulator mit dem weniger signifikanten Byte der Systemvariablen DF-CC geladen. DF-CC ist in den Adressen 16398 (400Eh) und 16399 zu finden. Der Akkumulator wird dann ins l-Register kopiert.

Ab Adresse 4086h wird dann das mehr signifikante Byte von DF-CC ins h-Register gebracht. Auch hier dient der Akkumulator wieder als Zwischenspeicher. Nach diesen Operationen ist die PRINT-Position im hl-Registerpaar enthalten.

Unter Adresse 408A wird dem c-Register der Inhalt der Speicherzelle zugewiesen, deren Adresse im hl-Registerpaar steht: Das c-Register erhält somit den Code des Zeichens, das die PRINT-Position beschreibt. Zwecks Rückmeldung nach Basic erhält das b-Register noch den Wert 00, so daß im bc-Registerpaar der erwünschte Wert enthalten ist.

Klaus Herklotz

*(Wird fortgesetzt)*

Aufruf
Lade l-Register über Akkumulator mit dem Inhalt von 400 Eh
Lade h-Register über Akkumulator mit dem Inhalt von 400 Eh
Lade c-Register mit dem Inhalt der Adresse des hl-Registers
Lade b-Register mit 00
Rücksprung
LET Q=PEEK (PEEK 16398+256 * PEEK 16399)

```
10 REM 000000000000
20 LET A$="3A0E406F3A0F
   40674E0600C9"
30 LET ADR=16514
40 FOR N=0 TO LEN A$-2 STEP 2
50 POKE ADR+INT (N/2),16*(CODE
   A$(N+1)-28)+CODE A$(N+2)-28
60 NEXT N
340 LET Q=USR 16514
```

⑤ **Maschinenroutine und Eingabeprogramm:** Der Ball wird schon deutlich schneller, wenn Zeile 340 des Basic-Programms unmittelbar auf Maschinenebene ausgeführt wird

| Adresse | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Befehl |
|---|---|---|---|---|---|
| 4082 | 3A | 0E | 40 | | ld a, (400E) |
| 4085 | 6F | | | | ld l, a |
| 4086 | 3A | 0F | 40 | | ld a, (400F) |
| 4089 | 67 | | | | ld h, a |
| 408A | 4E | | | | ld c, (hl) |
| 408B | 06 | 00 | | | ld b, 00 |
| 408D | C9 | | | | ret |

⑥ **Maschinenlisting:** Mit zwölf Bytes wird die Wirkung der Basic-Programmzeile 340 auf Maschinenebene realisiert

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 6: Auf Umwegen Adressieren

**Neben direkter und indirekter Adressierung gibt es eine noch indirektere Adressierung. Zu ihrer Demonstration dient ein kleiner Abstecher in die Arithmetik.**

Der Befehl zum indirekten Laden eines Registers läßt sich durch *ld r, (rp)* darstellen (siehe letzte Folge). Er legt fest, daß das Register r den Inhalt der 8-Bit-Speicherzelle rp erhält. Auch der umgekehrte Fall ist möglich: Eine Speicherzelle läßt sich mit einem Register-Inhalt laden. Der Befehl *ld (rp), r* ist dafür zuständig: Lade den Inhalt des Registers r in die Speicherzelle, die durch rp festgelegt wird. Der dazugehörige Befehlssatz mit den Codes ist nachfolgend aufgelistet.

| | | | |
|---|---|---|---|
| ld (NN), a | 32 | ld (hl), c | 71 |
| ld (bc), a | 02 | ld (hl), d | 72 |
| ld (de), a | 12 | ld (hl), e | 73 |
| ld (hl), a | 77 | ld (hl), h | 74 |
| ld (hl), b | 70 | ld (hl), l | 75 |

Wenn als Register r der Akkumulator verwendet wird, kann die Adresse der Speicherzelle auch direkt durch zwei Bytes angegeben werden. Es entsteht somit der Befehl *ld (NN), a*.

Als Übung versuchen wir, die Speicherzelle 16513 (4081h) mit 245 (F5h) zu laden: Dazu wird einfach der Akkumulator mit F5h geladen und der Inhalt anschließend in die Speicherzelle 4081h gebracht (*Bild 1*). Nach der Eingabe des Programmes und dem Aufruf durch LET Q=USR 16514 wird die REM-Zeile zur PRINT-Zeile. Eine Auflistung liefert den Beweis. Das ist eigentlich nicht überraschend, denn 245 ist der Code des PRINT-Befehls und 16513 die Adresse der Speicherzelle, in der der REM-Befehl stand.

Vergessen wir kurz einmal die Datentransportbefehle und fassen die Gruppe der arithmetischen Befehle ins Auge. Jetzt wird erstmal gerechnet.

### Ein Ausflug in die Arithmetik

Der einfachste Arithmetik-Befehl erhöht den Inhalt eines Registers oder Registerpaares um Eins. Das Kürzel *inc* dieser Operation steht für increase (engl: erhöhen, vergrößern). Nehmen wir an, im Akkumulator steht die Zahl 03. Nach der Operation *inc a* weist der Akkumulator die Zahl 04 auf.

Ein weiterer Befehl vermindert den Inhalt eines Registers oder Registerpaares um Eins. Sein Kürzel *dec* steht für decrease (engl.: vermindern). Im Akkumulator soll wieder die Zahl 03 stehen. Nach *dec a* ist der Akkumulatorinhalt auf 02 reduziert worden.

Nachfolgend sind alle Inkrementier- und Dekrementier-Befehle aufgelistet. Es können sämtliche Register und Registerpaare, sowie der Inhalt einer im hl-Registerpaar stehenden Adresse erhöht bzw. vermindert werden.

| | | | |
|---|---|---|---|
| inc a | 3C | dec a | 3D |
| inc b | 04 | dec b | 05 |
| inc c | 0C | dec c | 0D |
| inc d | 14 | dec d | 15 |
| inc e | 1C | dec e | 1D |
| inc h | 24 | dec h | 25 |
| inc l | 2C | dec l | 2D |
| inc (hl) | 34 | dec (hl) | 35 |
| inc bc | 03 | dec bc | 0B |
| inc de | 13 | dec de | 1B |
| inc hl | 23 | dec hl | 2B |

Das Maschinenprogramm aus *Bild 2* zeigt den Befehl *inc bc* in der Praxis (Start mit PRINT USR...). Dort wird zuerst das bc-Registerpaar mit 0003 geladen und dann der Inhalt um Eins erhöht. Nach dem Rücksprung wird somit 4 ausgegeben. Ersetzt man *inc bc* durch *dec bc*, so wird 2 ausgegeben.

### ZX 81 als Schnellschreiber

Viele ZX-81-Besitzer jammern über die Lahmheit ihres Rechners: „Man kann ja fast mitschreiben, wenn die Buchstaben nacheinander hingedruckt werden !“ So schlimm ist es zwar auch wieder nicht, aber dennoch werden wir dem ZX auf die Sprünge helfen.

Zum Problem: Es soll ein kleines Maschinenprogramm geschrieben werden, das „ZX-81“ auf den Bildschirm schreibt. Zuerst muß dazu die Adresse der linken, oberen Bildspeicherzelle ermittelt werden. Bei ihrer Ermittlung ist die Systemvariable D-FILE behilflich (vgl. Teil 2 in FS 11): Zuerst wird das höherwertige Byte von D-FILE ins h-Register und anschließend das niedrigerwertige Byte ins l-Register gebracht. Beide Male dient der Akkumulator als Zwischenspeicher.

Den weiteren Gang der Überlegung zeigt ein Flußdiagramm (*Bild 3*): Wenn

ADRESSE BYTES Z-80-ASSEMBLER

① **Speicherzellen belegen:** Die Speicherzelle 16513 erhält durch diese Maschinenroutine den Inhalt F5

② **Incrementierung:** Nach *inc bc* ist der Inhalt des Registerpaares bc nicht mehr 3, sondern 4

nun der Inhalt des hl-Registerpaares erhöht wird, beinhaltet es die Adresse der ersten Bildspeicherzelle. Dort hinein laden wir den Code von „Z“, nämlich 3Fh, und der erste Buchstabe ist schon am Bildschirm geschrieben. Der Inhalt des hl-Registerpaares wird anschließend erneut erhöht und der Code des nächsten Buchstaben geladen. So geht es weiter, bis auch der letzte Buchstabe geschrieben ist.

*Bild 4* zeigt das Maschinenlisting. Auf die gleiche Art und Weise läßt sich nun jeder beliebige Text mit ungemein hoher Geschwindigkeit auf den Bildschirm bringen. Doch Vorsicht! Das 33ste Zeichen in jeder Zeile muß NEW LINE sein.

Bei der direkten Adressierung ist der Operand – gemeint ist damit das zu ladende Byte – direkt im Befehl enthalten. Die Z-80-Assemblerschreibweise lautet dann so: *ld* r, r.

Bei der indirekten Adressierung ist der Operand nicht bekannt, sondern nur seine Adresse, die der Inhalt eines Registerpaares ist. Daraus resultiert die Schreibweise *ld r, (rp)*: Lade das Register r mit dem Inhalt der Adresse, die rp beschreibt.

## Indirekter geht's nicht

Es gibt jetzt aber noch eine quasi „hyperindirekte“ Adressierung: Man kennt dabei weder den Operanden noch seine Adresse, sondern nur die beiden aufeinanderfolgenden Adressen, in denen seine Adresse zu finden ist. Das klingt ziemlich kompliziert, aber genau den Fall hatten wir schon. Werfen wir noch einmal einen Blick auf Bild 4:

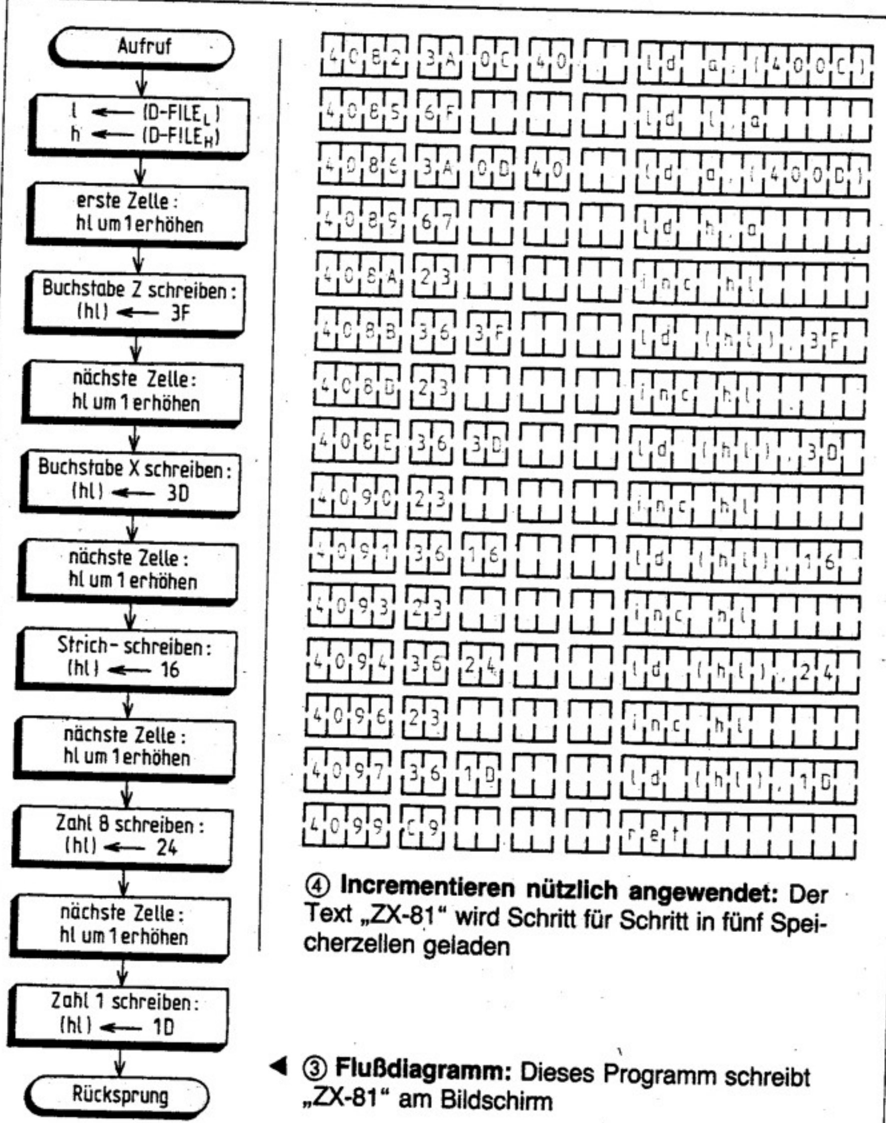
Von Adresse 4082h bis 4089h wird die Adresse der Bildspeichergrenze ermittelt. Die Adresse selbst ist noch unbekannt. Man kennt nur die beiden aufeinanderfolgenden Adressen, wo die beiden Adreß-Bytes zu finden sind. Um ein Registerpaar mit der tatsächlichen Adresse zu laden, sind dann vier Schritte mit insgesamt acht Bytes notwendig (zählen Sie selbst nach!). Ob dies wohl der Weisheit letzter Schluß ist? Sicher nicht, denn es geht viel kürzer!

Ausschlaggebend ist der Befehl *ld rp, (NN)*, oder speziell für das Programm aus Bild 4: *ld hl, (400Ch)*. Wie *Bild 5* zeigt, erfüllt der Befehl *ld hl, (400C)* eine Doppelfunktion. Einerseits wird das l-Register mit dem Inhalt der Speicherzelle 400Ch geladen. Auf der anderen Seite wird das h-Register mit dem Inhalt der nächsthöheren Speicherzelle, also mit dem von Adresse 400Dh, geladen.

Allgemein heißt das: Der Inhalt der angegebenen Speicherzelle NN wird ins niederwertige Register geladen. Danach sucht sich der Prozessor selbständig die nächste Speicherzelle NN+1 und lädt deren Inhalt ins höherwertige Register. *Bild 6* zeigt dazu den Befehlssatz.

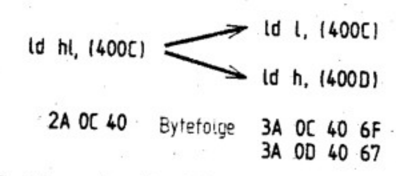
Jetzt läßt sich das Programm aus Bild 4 verkürzen, indem die ersten acht Bytes durch 2A, 0C, 40 (ld hl, (400C)) ersetzt werden. Wie Bild 6 weiterhin entnommen werden kann, ist diese Form der indirekten Adressierung auch umkehrbar: *ld (NN), rp* lädt den Inhalt des niedrigerwertigen Registers ($r_L$) in die Speicherzelle NN, sucht sich die nächste Speicherzelle NN+1 und lädt dort den Inhalt des höherwertigen Registers ($r_H$) hinein. Klaus Herklotz

*(Wird fortgesetzt)*




④ **Incrementieren nützlich angewendet:** Der Text „ZX-81“ wird Schritt für Schritt in fünf Speicherzellen geladen

◄ ③ **Flußdiagramm:** Dieses Programm schreibt „ZX-81“ am Bildschirm




⑤ **Eleganter Ladebefehl:** Mit nur drei Bytes wird der Inhalt von zwei Speicherzellen ins hl-Registerpaar geladen

| | |
|---|---|
| ld bc, (NN) ED 4B<br>ld de, (NN) ED 5B<br>ld hl, (NN) 2A | ld (NN), bc ED 43<br>ld (NN), de ED 53<br>ld (NN), hl 22 |
| $r_L$ ◄— (NN)<br>$r_H$ ◄— (NN+1) | (NN) ◄— $r_L$<br>(NN+1) ◄— $r_H$ |

⑥ **16-Bit-Ladebefehle zur indirekten Adressierung:** Wenn das hl-Registerpaar verwendet wird, sind nur drei Bytes notwendig

## Einführung in Z-80-Maschinensprache:

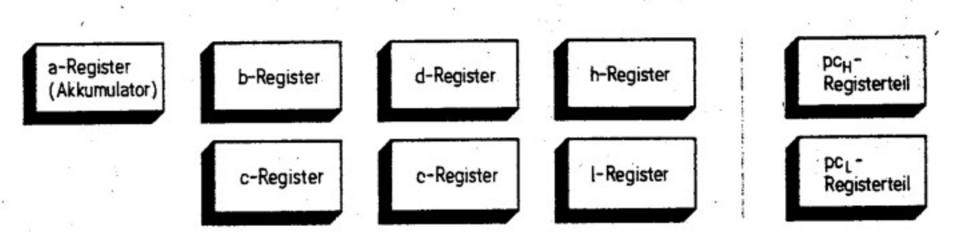
# Klartext für den ZX 81

## Teil 7: Jetzt wird im Kreis gesprungen

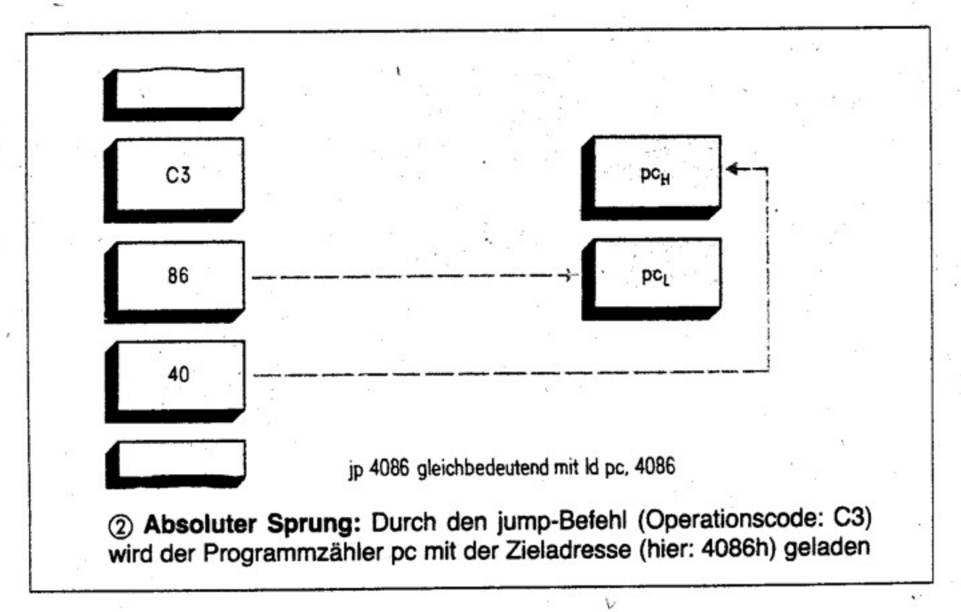
**Die Datentransportbefehle mit load wurden im letzten Teil abgeschlossen. Ab jetzt geht es um Sprungbefehle und um alles was dazu gehört.**

Wenn ein beliebiges Maschinenprogramm ausgeführt wird, dann werden die unter den einzelnen Adressen abgelegten Befehle nacheinander erledigt. Dies setzt voraus, daß der Prozessor immer weiß, welche Adresse als nächste an der Reihe ist.

Die Z-80-Entwickler haben deshalb der CPU einen Zähler spendiert, der stets die gerade aktuelle Adresse enthält und als Registerpaar realisiert ist. Die Rede ist vom Programmzähler oder pc-Register (program counter): Wann immer eine Adresse „abgearbeitet" worden ist, wird der Programmzähler inkrementiert (Wert um Eins erhöht). Wie *Bild 1* zeigt, erhöht sich damit unser Registervorrat auf neun Register.




① **Registervorrat:** Auch der 16 Bit umfassende Programmzähler besteht aus zwei 8-Bit-Registern (pc-Register)




② **Absoluter Sprung:** Durch den jump-Befehl (Operationscode: C3) wird der Programmzähler pc mit der Zieladresse (hier: 4086h) geladen

### Fast wie in Basic: der absolute Sprung

In Basic-Programmen folgt dem Schlüsselwort GOTO die Zeilennummer der Programmzeile, die als nächstes ausgeführt werden soll: Es erfolgt ein Sprung!

In Maschinensprache sind solche absoluten Sprünge ebenfalls programmierbar, nur läuft die Sache wieder unter einem anderen Namen ab: Hier kennt man den jump-Befehl mit dem Kürzel *jp*. Dieser Sprunganweisung folgt aber jetzt nicht die Programmzeile, sondern – wie in Maschinensprache üblich – die Adresse der Speicherzelle zu der gesprungen wird. So führt der Computer z. B. durch *jp 4082* als nächstes die in der Speicherzelle mit der Adresse 4082h stehende Anweisung aus.

Der Operationscode von *jp NN* ist „C3.. ..". Auch hier muß wieder das weniger bedeutende Adreß-Byte als erstes erscheinen. *Bild 2* verdeutlicht den Vorgang bei *jp NN*: Die beiden Adreß-Bytes werden in den Programmzähler geladen.

### Unser erstes Programm mit einem Sprungbefehl

Probieren wir den Sprungbefehl gleich an einem kurzen Maschinenprogramm aus. Recht viel Auswahl bleibt da kaum. Wir müssen uns, wie *Bild 3* zeigt, vorerst damit begnügen immer im Kreis zu springen.

Zuerst wird die Adresse der ersten Bildspeicherzelle ins hl-Registerpaar gebracht (4082h bis 4085h). In diese Zelle wird ein „X" geschrieben und gleich danach wieder gelöscht. Unter der Adresse 408A steht letztendlich der Operationscode des absoluten Sprunges gefolgt von der Zieladresse 4086h. Nach dem Aufruf dieses einfachen Programmes durch LET Q = USR 16514 flackert am linken oberen Bildschirmeck das X. Ein Nachteil läßt sich freilich nicht verschweigen: Nur durch Ziehen des Netzsteckers kann der Computer von der Flackerei erlöst werden!

Werfen wir noch kurz einen Blick zurück auf Bild 3. In der Z-80-Assemblerschreibweise ist die Zieladresse des Sprunges mit einem Label (Etikett)

### Was heißt Assembler?

Die Hexcodes der Maschinenbefehle lassen sich alles andere als leicht merken, so daß das Schreiben oder Analysieren längerer Maschinenprogramme sehr mühsam würde. Leichter geht das mit den symbolischen Abkürzungen (Mnemonics) der Maschinenbefehle (z. B. ret für Return). Ein mit symbolischen Abkürzungen geschriebenes Programm versteht der Prozessor freilich ebensowenig wie ein Basic-Programm, wenn ihm die Befehle nicht in Hexcodes übersetzt werden. In Basic hat diese Aufgabe z. B. ein Interpreter, wogegen Mnemo-Programme von einem *Assembler* übersetzt werden (der Assembler selbst ist meist ein Hexcode-Programm). Ein *Disassembler* übersetzt Hexcodes in die entsprechenden Mnemo-Abkürzungen (zur Programmanalyse). Programmieren in Assembler heißt, daß hier mit Mnemonics gearbeitet wird.

versehen. Dabei wird so vorgegangen, daß man für die Label-Spalte einfach einen passenden Namen erfindet (z. B. FLACK für flackern). Dann braucht auch der Befehlscode des Sprunges (bei uns in Adresse 408A) nicht von der Zieladresse selbst, sondern lediglich von ihrem Etikett gefolgt werden. Dies bringt schon fürs erste ein hohes Maß an Übersichtlichkeit.

## Der „Stapel" als Datenspeicher

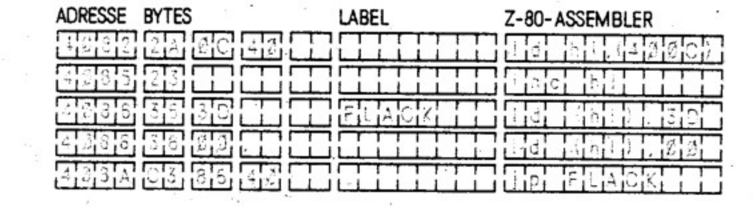
Jeder noch so billige Taschenrechner hat einen Zahlenwert-Speicher. Mit speziellen Tasten können Zahlen dort eingespeichert und zu einem späteren Zeitpunkt wieder abgerufen werden. Da unser Prozessor bislang nur sieben Register (ohne Programmzähler) zur Verfügung stellt, wäre ein solcher Speicher eine starke Sache.

Im Z 80 ist der Speicher als „Stapel" (engl.: Stack) gebaut. Wie *Bild* 4 zeigt, befindet sich am einen Ende des RAM-Bereichs ein beliebiges Programm. Je länger das Programm ist, desto weiter reicht es in den RAM-Bereich hinein.

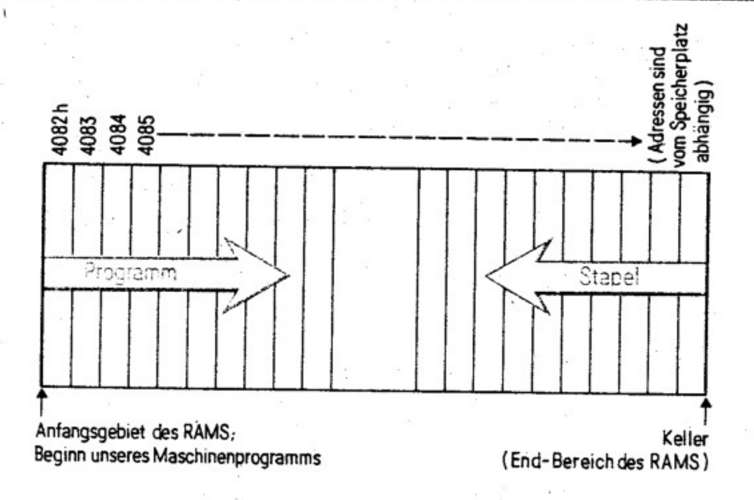
Fast ganz am anderen Ende des RAM-Bereichs befindet sich der „Keller" und zwar der Keller des Stapel-Speichers. Auf diesen Keller können wir nun Inhalte von Registerpaaren speichern. Dies geschieht wie bei einem Stapel Spielkarten: Die Karte, die als letzte auf den Stapel gelegt wurde muß man später auch als erste wieder wegnehmen. Genaueres darüber im nächsten Teil.

Klaus Herklotz

*(Wird fortgesetzt)*




③ **Programm ohne Ende:** Durch eine mit jump gebildete Endlosschleife flackert ein Buchstabe im „Maschinensprachtempo"




④ **Grenzen des RAM-Speichers:** Das Maschinenprogramm steht am Beginn des RAMs bei niedrigen Adressen. Der Stapel wächst dem Programm von der Obergrenze des RAMs entgegen. Für den Stapel ist die RAM-Obergrenze der Keller

ZX-81-Softwaretip:

## Bremse für „Autostart"

Manche ZX-81-Programme im Handel haben die Eigenschaft, nach dem Laden von Kassette automatisch zu starten und sich dann nicht mehr unterbrechen zu lassen. Dies stört, wenn man so ein Programm näher untersuchen, kopieren oder abspeichern möchte. Das folgende Hilfsprogramm (Maschinencode) ermöglicht es, solche Programme so zu laden, daß sie nicht automatisch anlaufen:

```
1 REM 1234567890A
2 FOR A=16514 TO 16524
3 INPUT X
4 POKE A,X
5 NEXT A
```

Nach dem Start durch RUN sind die elf Zahlen 33, 102, 0, 229, 46, 8, 229, 55, 195, 67, 3 einzugeben und danach die folgenden Programmzeilen.

```
2 REM SPECIAL LOAD
3 FAST
4 RUN USR 16514
```

Jetzt kann Zeile 5 gelöscht und das Programm auf einer Kassette gespeichert werden. Nach dem Start des Programms durch RUN wird vom Maschinenprogramm die LOAD-Routine im ROM aufgerufen (Adresse: 835) und dadurch der Befehl LOAD " " ausgeführt. Nach erfolgreicher Beendigung des Ladevorgangs wird grundsätzlich mit Meldungscode 9 gestoppt. Hierfür verantwortlich sind spezielle Daten, die vor dem eigentlichen LOAD auf den Z-80-Stack gelegt worden sind.

Michael Schramm

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 8: Der Umgang mit dem „Stapel“

**Daß der Stapel ein spezieller Bereich im RAM ist, wo man die Inhalte der Z-80-Register ablegen kann, wurde schon angesprochen. Jetzt kommt es darauf an, ihn zu Nutzen.**

In der Z-80-Maschinensprache werden Inhalte von Registerpaaren durch den push-Befehl gespeichert. So wirft z. B. *push bc* eine Kopie des bc-Registerpaares auf den Stapel, speichert also den Inhalt des bc-Registerpaars. Der Vorgang ist damit zu vergleichen, daß eine Karte auf einen Stoß mit Spielkarten gelegt wird.

Wie *Bild 1* zu entnehmen ist, geschieht das Einspeichern von Daten aus Registerpaaren in zwei Schritten. Zuerst wird das höherwertige Register und dann das niedrigerwertige Register auf den Stapel geworfen. Nun wissen wir also, wie solche Daten gespeichert werden. Es fehlt nur noch die Möglichkeit, die Daten wieder abzurufen.

### So werden Daten vom Stapel „abgehoben“

Die Möglichkeit, dem Stapel Daten zu entnehmen bietet der pop-Befehl. So holt sich z. B. *pop de* die obersten beiden Bytes vom Stapel und lädt sie ins de-Registerpaar. Wenn wir wieder an unser Kartenspiel denken, heißt das, daß wir die oberste Karte abheben.

Auch das Abrufen von Daten geschieht wieder in zwei Schritten (*Bild 2*): Zuerst wird das oberste Byte vom Stapel ins niedrigerwertige Register und dann das nächste Byte ins höherwertige Register geladen.

### Der Stapel hilft uns beim Kopieren

Lösen wir jetzt ein altbekanntes Problem auf neue Weise: Es soll das hl-Registerpaar mit 0001 geladen und dann ins bc-Registerpaar kopiert werden. Der Ladevorgang des hl-Registerpaares soll dabei unverändert bleiben. Wie aus dem Maschinenlisting (*Bild 3*) ersichtlich, speichern wir dazu das hl-Registerpaar auf dem Stapel. Mit dem nächsten Schritt wird dieses Datenpaar durch *pop bc* ins bc-Registerpaar geladen. Nach dem Aufruf des Maschinenprogrammes mit PRINT USR 16514 wird wie erwartet 1 ausgegeben.

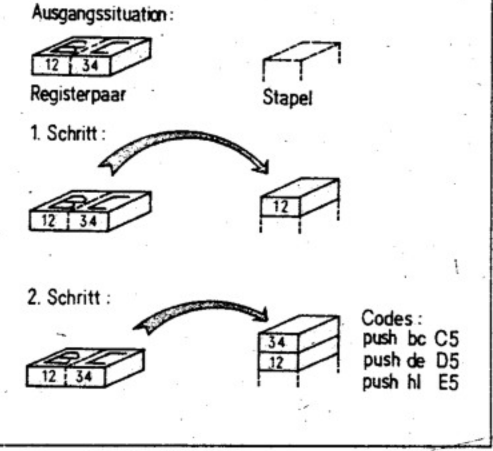
### Eine Gedächtnisstütze für den Prozessor

Genauso wie der Prozessor immer die gerade bearbeitete Adresse bei der Programmausführung kennen muß, so muß er auch wissen, bei welcher Adresse das obere Ende des Stapels liegt. Denn wie soll er sonst dort Daten aus Registerpaaren ablegen? Aus gutem Grund wurde deshalb ein „Stapelzeiger“ installiert.

Der Stapelzeiger ist wieder als Registerpaar vorhanden, ähnlich dem Programmzähler. Gemeint ist der Stackpointer (Stapelzeiger) oder das sp-Register.

Immer wenn wir Daten auf dem Stapel ablegen (z. B. durch push), wird das sp-Register um 2 erniedrigt. Wem das nicht klar ist, der betrachte noch einmal Bild 4 aus Teil 7: Immer wenn Daten auf dem Stapel abgelegt werden, vergrößert das zwar den Stapel selbst, jedoch verringern sich die Werte der Adressen!

Wenn dem Stapel nun Daten entnommen werden (z. B. durch *pop*), dann wird das sp-Register um 2 erhöht. Beachtlich bei dem Ganzen ist, daß der Stapel selbst dabei nicht kleiner wird (Bild 2): Daten, die einmal eingespeichert wurden, werden nicht mehr gelöscht; es sei denn sie werden durch erneutes Einspeichern überschrieben. Diesem Umstand darf jedoch wenig Aufmerksamkeit beigemessen werden, weil er für den Programmieralltag nebensächlich ist.




① **Daten stapeln:** Die Ausgangssituation zeigt das bc-Registerpaar (Inhalt 1234) und das obere Ende des Stapels. Im ersten Schritt wird der Inhalt des höherwertigen Registers auf den Stapel gelegt und im zweiten Schritt der Inhalt des niederwertigen Registers

## Der Stapel hilft bei Unterprogrammen

Wenn ein und dieselben Programmteile in einem Hauptprogramm mehrmals ausgeführt werden müssen, so empfiehlt es sich, Unterprogramme zu verwenden. In Basic werden Unterprogramme durch den GOSUB-Befehl mit nachfolgender Zeilennummer aufgerufen. Am Ende des Unterprogrammes steht dann der RETURN-Befehl, der einen Rücksprung ins Hauptprogramm zur Folge hat. In der Z-80-Maschinensprache kann die gleiche Wirkung mit den Befehlen *call NN* und *ret* (für return) erreicht werden.

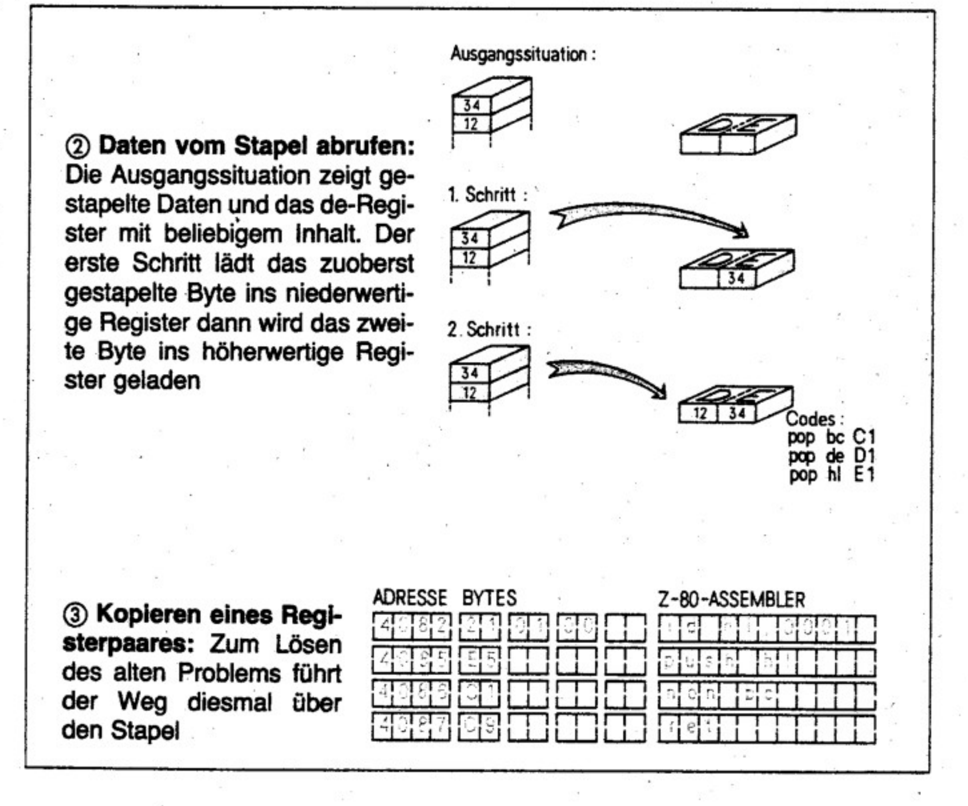
Die Eingabe *call NN* bewirkt dasselbe wie *jp NN*, außer daß sich der Computer den Programmzählerstand vor dem Absprung (Rücksprungadresse) merkt. Die Rücksprungadresse wird dabei einfach oben auf den Stapel gelegt (*Bild* 4).

Der Befehl *ret* nimmt die obersten beiden Bytes des Stapels ab und lädt sie in den Programmzähler: Es erfolgt der Rücksprung. Der hexadezimale Operationscode für *call NN* lautet „CD .. ..“; für *ret* ist er bekanntlich „C9“.

Zum Abschluß dieses Teils überlegen wir uns, wie ein Maschinenprogramm aussehen muß, das dreimal nebeneinander „ZX-81“ auf den Bildschirm schreibt. Dazu nehmen wir das Maschinenlisting zum schnellen Drucken aus Teil 6 und ergänzen es mit Hilfe von Unterprogrammtechniken. *Bild* 5 zeigt einen Lösungsvorschlag, der aber nicht im einzelnen besprochen werden soll, weil das Listing teilweise bekannt und außerdem leicht zu verstehen ist.

Klaus Herklotz

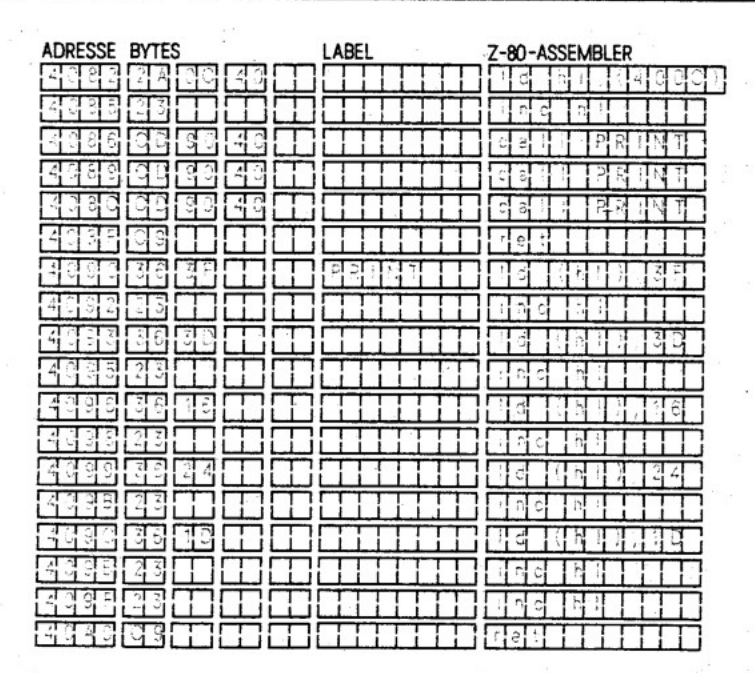
(*Wird fortgesetzt*)




② **Daten vom Stapel abrufen:** Die Ausgangssituation zeigt gestapelte Daten und das de-Register mit beliebigem Inhalt. Der erste Schritt lädt das zuoberst gestapelte Byte ins niederwertige Register dann wird das zweite Byte ins höherwertige Register geladen

| ADRESSE | BYTES | | | | Z-80-ASSEMBLER |
|---|---|---|---|---|---|
| 4082 | 21 | 01 | 30 | | ld hl,3001 |
| 4085 | E5 | | | | push hl |
| 4086 | C1 | | | | pop bc |
| 4087 | C9 | | | | ret |

③ **Kopieren eines Registerpaares:** Zum Lösen des alten Problems führt der Weg diesmal über den Stapel



| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | 2A | 0C | 40 | | | ld hl,(400C) |
| 4085 | 23 | | | | | inc hl |
| 4086 | CD | 90 | 40 | | | call PRINT |
| 4089 | CD | 90 | 40 | | | call PRINT |
| 408C | CD | 90 | 40 | | | call PRINT |
| 408F | C9 | | | | | ret |
| 4090 | 36 | 3F | | | PRINT | ld (hl),3F |
| 4092 | 23 | | | | | inc hl |
| 4093 | 36 | 3D | | | | ld (hl),3D |
| 4095 | 23 | | | | | inc hl |
| 4096 | 36 | 1E | | | | ld (hl),1E |
| 4098 | 23 | | | | | inc hl |
| 4099 | 36 | 24 | | | | ld (hl),24 |
| 409B | 23 | | | | | inc hl |
| 409C | 36 | 1D | | | | ld (hl),1D |
| 409E | 23 | | | | | inc hl |
| 409F | 23 | | | | | inc hl |
| 40A0 | C9 | | | | | ret |

⑤ **Anwendung eines Unterprogramms:** Dieses Programm sorgt dafür, daß der Text „ZX-81“ dreimal auf den Bildschirm geschrieben wird

```
call NN ──→ push pc
        ──→ jp NN

ret     ──→ pop pc
```

④ **Unterprogramm-Befehle:** Die Rücksprungadresse liegt auf dem Stapel. Deshalb müssen innerhalb von Unterprogrammen Daten, die mit *push* auf den Stapel gelegt wurden, unbedingt durch *pop* wieder abgehoben werden, damit der Z 80 die Rücksprungadresse findet

Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 9: Zugriff auf einzelne Bits

**Bit ist die Abkürzung von binary digit (Zweierschritt). Es ist das kleinste Informationselement im Computer und kennt nur die Signalzustände „0“ (keine Spannung) und „1“ (volle Spannung). Auf diese Bits greifen wir jetzt zu.**

Um die Vorgänge im Computer einigermaßen überblicken zu können, haben Mikroprozessor-Entwickler einzelne Bits zu Gruppen zusammengefaßt. Bei der Z-80-CPU sind es 8-Bit-Gruppen, was sich an der 8-Bit-Breite von Speicherzellen, Datenbus und Registern feststellen läßt.

Wenn das c-Register mit der Hex-Zahl 03 geladen wird (Z-80-Assembler: *ld c, 03*), so bedeutet dies, daß im c-Register nur Bit 0 und Bit 1 den Signalzustand „1“ aufweisen; Bit 2 bis Bit 7 sind gelöscht (*Bild 1*).

Die Z-80-Maschinensprache bietet außer dieser direkten Adressierung noch die Möglichkeit, jedes Bit einzeln anzusteuern (bitweise Adressierung). Wie *Bild 2* zeigt, können Bits aus beliebigen Registern wahlweise gesetzt (Signalszustand „1“) oder zurückgesetzt (Signalzustand „0“) werden.

*set b, r* ordnet dem Bit b des Registers r den Signalzustand „1“ zu, während *res b, r* dem Bit b des Registers r den Signalzustand „0“ zuordnet.

C-Register

| 0 | 3 |
|---|---|

| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|
| Bit 7 ≙ $2^7$ | Bit 6 ≙ $2^6$ | Bit 5 ≙ $2^5$ | Bit 4 ≙ $2^4$ | Bit 3 ≙ $2^3$ | Bit 2 ≙ $2^2$ | Bit 1 ≙ $2^1$ | Bit 0 ≙ $2^0$ |

① **Blick ins Detail:** Diese Signalzustände haben die Bits des c-Registers, wenn es mit 03h geladen ist. 03h = $2^0+2^1$, deshalb sind nur Bit 0 und Bit 1 gesetzt

Die bitweise Adressierung soll wieder praktisch erprobt werden. Versuchen wir dazu das bc-Registerpaar mit 03 zu laden (*Bild 3*). Zuerst werden auf herkömmliche Art alle Bits des bc-Registerpaares zurückgesetzt (4082h). Danach werden Bit 0 (4085h) und auch Bit 1 des c-Registers gesetzt (4087h). Den Abschluß des Maschinenprogrammes bildet wie gewöhnlich der Rücksprung ins Basic. Nach dem Einpoken und dem Aufruf durch PRINT USR 16514 wird erwartungsgemäß die Zahl 3 auf den Bildschirm geschrieben.

### Eigentlich ganz logisch: Logische Verknüpfungen

Ein Programm soll vom Kassettenrecorder in den ZX 81 geladen werden: Man kann den Ladevorgang z. B. nur dann erfolgreich beenden, wenn die Stromversorgung beider Geräte einwandfrei war und wenn die Verbindungskabel richtig angeschlossen wurden.

Beide Aussagen sind durch *und* miteinander verknüpft. Man bezeichnet das daher als UND-Verknüpfung (engl: AND): Das Ergebnis fällt nur dann positiv aus, wenn alle Teilaussagen positiv sind. Auf den Mikroprozessor übertragen bedeutet das: Nur wenn alle Eingänge den Signalzustand „1“ aufweisen, darf auch der Ausgang den Signalzustand „1“ erhalten (*Bild 4*).

Im Gegensatz dazu soll ein Programm erfolgreich ausgeführt werden: Das ist z. B. nur möglich, wenn man die einzelnen Programmzeilen richtig eingetippt hat, oder wenn das Programm richtig geladen wurde.

Die Aussagen sind jetzt durch *oder* miteinander verknüpft. Es liegt eine ODER-Verknüpfung vor (engl.: OR): Das Ergebnis fällt schon positiv aus, sobald nur eine Teilaussage positiv ist. Für unseren Prozessor gilt wieder: Sobald ein Eingang den Signalzustand „1“ aufweist, soll auch der Ausgang den Signalzustand „1“ erhalten.

Schließlich soll ein Programm durch Drücken der Tasten P oder Q unterbrochen werden. Es wird nur dann unterbrochen, wenn eine der beiden Tasten

ADRESSE BYTES Z-80 ASSEMBLER

③ **Laden des bc-Registerpaares:** Durch bitweises Adressieren läßt sich z. B. die Zahl 0003h (umständlich) laden

② **Bitweises Adressieren:** Der Z-80-Befehlsvorrat erlaubt es, Bits einzeln zu setzen oder rücksetzen (set, res). Jeder der Befehle erfordert zwei 8-Bit-Speicherzellen, da stets CBh vorangesetzt ist

| Register r: | a | b | c | d | e | h | l | (hl) |
|---|---|---|---|---|---|---|---|---|
| set 0, r | CBC7 | CBC0 | CBC1 | CBC2 | CBC3 | CBC4 | CBC5 | CBC6 |
| set 1, r | CBCF | CBC8 | CBC9 | CBCA | CBCB | CBCC | CBCD | CBCE |
| set 2, r | CBD7 | CBD0 | CBD1 | CBD2 | CBD3 | CBD4 | CBD5 | CBD6 |
| set 3, r | CBDF | CBD8 | CBD9 | CBDA | CBDB | CBDC | CBDD | CBDE |
| set 4, r | CBE7 | CBE0 | CBE1 | CBE2 | CBE3 | CBE4 | CBE5 | CBE6 |
| set 5, r | CBEF | CBE8 | CBE9 | CBEA | CBEB | CBEC | CBED | CBEE |
| set 6, r | CBF7 | CBF0 | CBF1 | CBF2 | CBF3 | CBF4 | CBF5 | CBF6 |
| set 7, r | CBFF | CBF8 | CBF9 | CBFA | CBFB | CBFC | CBFD | CBFE |
| res 0, r | CB87 | CB80 | CB81 | CB82 | CB83 | CB84 | CB85 | CB86 |
| res 1, r | CB8F | CB88 | CB89 | CB8A | CB8B | CB8C | CB8D | CB8E |
| res 2, r | CB97 | CB90 | CB91 | CB92 | CB93 | CB94 | CB95 | CB96 |
| res 3, r | CB9F | CB98 | CB99 | CB9A | CB9B | CB9C | CB9D | CB9E |
| res 4, r | CBA7 | CBA0 | CBA1 | CBA2 | CBA3 | CBA4 | CBA5 | CBA6 |
| res 5, r | CBAF | CBA8 | CBA9 | CBAA | CBAB | CBAC | CBAD | CBAE |
| res 6, r | CBB7 | CBB0 | CBB1 | CBB2 | CBB3 | CBB4 | CBB5 | CBB6 |
| res 7, r | CBBF | CBB8 | CBB9 | CBBA | CBBB | CBBC | CBBD | CBBE |

**UND-Verknüpfung**

Symbolschreibweise: E1, E2 → A

Funktionstabelle

| Eingänge | | Ausgang |
|---|---|---|
| $E_1$ | $E_2$ | A |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**ODER-Verknüpfung**

Symbolschreibweise: E1, E2 → A

Funktionstabelle

| Eingänge | | Ausgang |
|---|---|---|
| $E_1$ | $E_2$ | A |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Exklusiv-ODER-Verknüpfung**

Symbolschreibweise: E1, E2 → A

Funktionstabelle

| Eingänge | | Ausgang |
|---|---|---|
| $E_1$ | $E_2$ | A |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

④ **Logische Verknüpfungen:** Vom Signalzustand an den Eingängen hängt der Signalzustand am Ausgang ab. Die Z-80-CPU simuliert jeweils acht solcher Verknüpfungen auf einmal

allein gedrückt wird; ansonsten passiert nichts.

Hier sind die Aussagen durch eine ODER-ähnliche Verknüpfung verbunden. Dieser Fall wird als Exklusiv-ODER-Verknüpfung bezeichnet (engl.: EXOR): Das Ergebnis fällt nur dann positiv aus, wenn die Signalzustände an beiden Eingängen verschieden sind. Für Mikroprozessoren gilt: Sobald beide Eingänge verschiedene Signalzustände aufweisen, erhält der Ausgang den Signalzustand „1". Was nun bei der Z-80-CPU die jeweiligen Ein- bzw. Ausgänge sind und wie sie verknüpft werden, wird im nächsten Teil verraten.

Klaus Herklotz

*Wird fortgesetzt*

## Softwaretip:

# Texteingabe

Ohne viel zu probieren, ermöglicht folgende Eingabemethode für Texte in Verbindung mit PRINT-Anweisungen auf Anhieb zufriedenstellende Ergebnisse. Es gilt lediglich das tatsächliche Zeilenende beim Ausführen der PRINT-Anweisung zu ermitteln, da dieses nicht mit dem Zeilenende beim Schreiben der Anweisung übereinstimmt (siehe FS 11/83, Seite 80).

Um das tatsächliche Zeilenende bereits beim Schreiben der PRINT-Anweisung zu kennen, ist einfach nach den ersten Anführungszeichen ein senkrechter Strich am Bildschirm zu ziehen (Filzstift); er markiert das Zeilenende. Bei PRINT-AT-Anweisungen ist zusätzlich die durch AT verursachte Verschiebung zu berücksichtigen. Lautet die Eingabe z. B. 10 PRINT AT 1,10;"XXX..." dann muß der Strich zehn Spalten links vom ersten Anführungszeichen gezogen werden. Nur wenn eine PRINT-Anweisung mit einem Strichpunkt abgeschlossen wird und danach eine PRINT-Anweisung ohne Positionierung erfolgt, dann versagt die Methode.

Reinhold Woehler

## Klartext im Detail:

# Von 8-Bit- und 1-Byte-Befehlen

In Teil 5 der Serie sind sie zum ersten Male aufgetaucht: 8-Bit-, 16-Bit-, 1-Byte- und 2-Byte-Befehle. Da 8 Bit auf jeden Fall 1 Byte ergeben, ist auf Anhieb nicht einzusehen, warum ein und derselbe Befehl in beiden Schreibweisen vorkommt (Schusselei einmal ausgeschlossen). Um es vorwegzunehmen: Dieser kleine Trick hilft Verwechslungen zu vermeiden.

Eigentlich müßte der Befehl *ld a, N* als 1-Byte-Befehl bezeichnet werden, weil der Akkumulator mit einer Ein-Byte-Zahl geladen wird. Im Prinzip ist das richtig, aber nicht sehr eindeutig. Denn bedenken Sie: Man könnte den Befehl *ld a, N* auch als 2-Byte-Befehl bezeichnen, da zu seiner Ausführung zwei Bytes benötigt werden.

In der gängigen Literatur – an die auch wir uns halten wollen – wird deshalb ein Unterschied gemacht: Soll mitgeteilt werden, daß die Länge des Befehls (also die Gesamtzahl der benötigten Bytes) gemeint ist, so erfolgt die Angabe in Byte bzw. Wort: *ld a, N* ist somit ein 2-Byte- bzw. 2-Wort-Befehl. Will man dagegen bekunden, welche Länge der zu ladende Wert hat, so erfolgt die Angabe in Bit: *ld a, N* ist dann ein 8-Bit-Befehl; *ld bc, N* ein 16-Bit-Befehl.

Klaus Herklotz

### Vergleichen Sie Basic mit Maschinensprache

Was eigentlich unterscheidet das Programmieren in Maschinensprache von dem in Basic? Machen Sie sich die Unterschiede ruhig einmal klar, um bei Fragen sattelfest Rede und Antwort stehen zu können.

Ein Programm das in Basic geschrieben ist, wird durch Eintippen von Programmzeilen eingegeben. Ein Maschinenprogramm dagegen wird durch Poken von Speicherzellen eingegeben. Es wurde im Rahmen der Serie „Klartext für den ZX 81" vereinbart, daß es sich dabei um die Adressen der Zeichen einer REM-Zeile am Anfang des Arbeitsspeichers handelt.

Ein Basic-Programm wird durch den RUN-Befehl zum Laufen gebracht, worauf die einzelnen Programmzeilen nacheinander ausgeführt werden. Ein Maschinenprogramm wiederum wird von der Basic-Ebene durch die USR-Funktion abgerufen. Dann werden die einzelnen Speicherzellen hintereinander „erledigt".

In Basic sind Befehle durch Schlüsselwörter (z. B. PRINT) definiert. Rechnen geschieht mit Variablen (z. B. LET A=3 oder LET B=C). In Maschinensprache sind Befehle durch Befehlscodes (Zahlen) festgelegt. Rechnen erfolgt in Registern (z. B. ld a, 03 oder ld b, c). Beim Rücksprung nach Basic wird beim ZX 81 das bc-Registerpaar ausgedruckt.



Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 10: Logik im Computer

**Logische Verknüpfungen sind für den Computer kein Problem, wenn es sich um AND-, OR- oder EXOR-Funktionen handelt. Auch das Addieren und Subtrahieren fußt auf dieser Logik.**

Der Z-80-Befehlsvorrat erlaubt es, logische Verknüpfungen zwischen Akkumulator- und Registerinhalten zu programmieren. Wie aus *Bild 1* ersichtlich ist werden die entsprechenden Bits des Akkumulators paarweise mit den Bits des gewählten Registers verknüpft und das Ergebnis dann im Akkumulator abgelegt.

Der Befehl *and b* z. B. verknüpft die Bits des Akkumulators mit den Bits des b-Registers gleicher Wertigkeit durch ein logisches UND, d. h. das Ergebnisbit wird nur dann gesetzt, wenn das Bit des Akkumulators und das Bit des b-Registers gesetzt ist. Ähnlich verknüpft *or c* die Bits des Akkumulators mit den Bits des c-Registers durch ein logisches ODER, während *xor d* die Bits durch ein logisches Exklusiv-ODER verknüpft.

### Logische Beispiele

Die drei Verknüpfungsarten wollen wir sofort erproben (*Bild 2*). In jedem der drei Programme wird zuerst die Ausgangssituation geschaffen: Der Akkumulator wird mit 03 geladen (Adresse 4082h) und das gewählte Register mit 05 (4084h). Anschließend folgt eine der Verknüpfungsarten (4086h). Zwecks Rücksprung wird letztendlich das Ergebnis der Verknüpfung ins bc-Registerpaar gebracht (4087h bis 4089h). Nach dem Aufruf der Programme mit PRINT USR 16514 wird nach der UND-Verknüpfung „1", nach der ODER-Verknüpfung „7" und nach der Exklusiv-ODER-Verknüpfung „6" ausgegeben. Wem das nicht klar ist, dem hilft ein Rückblick auf Bild 1: Alle Bits werden paarweise verknüpft!

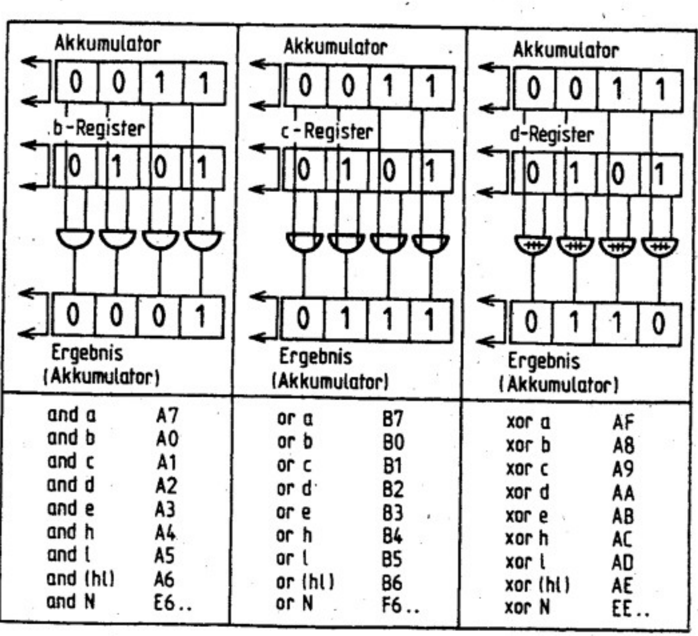
Ein wenig schleierhaft scheinen die Befehle *or a*, *and a* und *xor a* zu sein (Bild 1). Denn welchen Sinn kann es haben, den Akkumulator mit sich selbst zu verknüpfen? Des Rätsels Lösung läßt sich am Beispiel von *xor a* zeigen:

Bei einer Exklusiv-ODER-Verknüpfung wird das Ergebnisbit immer nur dann 1, wenn die Signalzustände beider Teileingänge verschieden sind. Offensichtlich sind die beiden Teileingänge bei *xor a* Bits aus ein und demselben Register (hier: Akkumulator). Das Ergebnisbit ist also unabhängig vom Inhalt des Akkumulators immer 0! Somit ersetzt *xor a* platzsparend den 2-Byte-Befehl *ld a, 00*.

### Arithmetik baut auf Logik

Logische Verknüpfungen ermöglichen dem Computer einfache Rechenoperationen. Das sind im einzelnen Addition und Subtraktion. Dabei werden die einzelnen Bits ziemlich kompliziert verknüpft, was uns aber weiter nicht kümmert.

Wie bei logischen Befehlen sind auch bei arithmetischen Rechenaufga-




① **Verknüpfungen in der CPU:** Das Ergebnis steht im Akkumulator. Der Akkumulatorinhalt läßt sich auch mit einer 8-Bit-Zahl oder mit dem Inhalt einer durch das hl-Registerpaar beschriebenen Speicherzelle verknüpfen

ben der Akkumulator und ein frei wählbares Register die Operanden. So addiert z. B. *add a, b* den Inhalt des b-Registers zum Inhalt des Akkumulators. Ähnlich verhält sich die Angelegenheit bei *sub a, b:* Der Inhalt des b-Registers wird vom Inhalt des Akkumulators subtrahiert. Das Ergebnis der Rechenoperation erhält beide Male der Akkumulator.

Arithmetische Befehle gibt es in 8-Bit- und in 16-Bit-Ausführung (*Bild 3*). Bei den arithmetischen 16-Bit-Befehlen ist das hl-Registerpaar Ausgangspunkt. Der Befehl *add hl, bc* z. B. addiert das bc- zum hl-Registerpaar.

Versuchen wir jetzt, ein Maschinenprogramm zu schreiben, das ein A in die zehnte Spalte der zehnten Zeile schreibt. Einen Lösungsvorschlag zeigt das Listing aus *Bild 4*.

Wie gewöhnlich erhält das hl-Registerpaar die untere Grenze des Bildspeicherbereichs zugewiesen (Adresse 4082h). Die anzusteuernde Bildspeicherzelle ist die 341ste ihrer Art. Deshalb wird das bc-Registerpaar mit 155h geladen (4085h) und zum Inhalt des hl-Registerpaars addiert (4088h). Letztendlich wird der Buchstabe auf den Bildschirm gebracht (4089h).

8 Bit breite arithmetische und logische Befehle ermöglichen es dem Programmierer, vielseitige Verknüpfungen zwischen Akkumulator und Registern durchzuführen. 16-Bit-Befehle gibt es nur zur Arithmetik. Wie wir später sehen werden, sind die Anwendungsbereiche der arithmetisch/logischen Befehle viel größer, als dies jetzt den Anschein hat.

Klaus Herklotz

*(wird fortgesetzt)*

```
ADRESSE  BYTES         Z-80-ASSEMBLER
4082     3E 03         ld a,03
4084     06 05         ld b,05
4086     A0            and b
4087     06 00         ld b,00
4089     4F            ld c,a
408A     C9            ret

4082     3E 03         ld a,03
4084     0E 05         ld c,05
4086     B1            or c
4087     06 00         ld b,00
4089     4F            ld c,a
408A     C9            ret

4082     3E 03         ld a,03
4084     16 05         ld d,05
4086     AA            xor d
4087     06 00         ld b,00
4089     4F            ld c,a
408A     C9            ret
```

② **Logik in der Praxis:** Wer die Ergebnisse verstehen will, dem bleibt das binäre Zahlensystem (Sinclair-Handbuch, Kapitel 24) nicht erspart. Beispiel: 3 xor 5 = 00 000 011 xor 00 000 101 = 00 000 110 = 6h

| | | | | | |
|---|---|---|---|---|---|
| add a, a | 87 | sub a, a | 97 | add hl, bc | 09 |
| add a, b | 80 | sub a, b | 90 | add hl, de | 19 |
| add a, c | 81 | sub a, c | 91 | add hl, hl | 29 |
| add a, d | 82 | sub a, d | 92 | add hl, sp | 39 |
| add a, e | 83 | sub a, e | 93 | | |
| add a, h | 84 | sub a, h | 94 | | |
| add a, l | 85 | sub a, l | 95 | | |
| add a, (hl) | 86 | sub a, (hl) | 96 | | |
| add a, N | C6 | sub a, N | D6 | | |

③ **Arithmetische Befehle:** Beim Addieren von 16-Bit-Zahlen läßt sich auch der Inhalt des Stapelzeigers zum hl-Registerpaar addieren

```
ADRESSE  BYTES         Z-80-ASSEMBLER
4082     2A 0C 40      ld hl,(400C)
4085     01 55 01      ld bc,0155
4088     09            add hl,bc
4089     36 26         ld (hl),26
408B     C9            ret
```

③ **Arithmetische Befehle:** Beim Addieren von 16-Bit-Zahlen läßt sich auch der Inhalt des Stapelzeigers zum hl-Registerpaar addieren

Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 11: Die CPU zeigt Flagge

**Um die „Flaggen-Zeichen“ des Computers verstehen zu können, machen wir eine kurze Reise in die Vergangenheit.**

Ein Segelschiff läuft einen Hafen an, und der Kapitän will den Hafenarbeitern schon von weitem signalisieren, ob sich Ladung auf dem Schiff befindet. Sie haben deshalb vereinbart, daß der Kapitän eine rote Flagge (engl.: Flag) setzt, wenn er *keine* Ladung an Bord hat. Sobald sich Ladung an Bord befindet, ist die Flagge nicht gesetzt. Für einen Hafenarbeiter ist es damit ein leichtes festzustellen, ob er mit Ladung rechnen muß.

### Die Bedeutung des Zero-Flags

Der Z-80-Prozessor hat keine rote Flagge, sondern ein sogenanntes Zero-Flag (Null-Flagge). Das Zero-Flag wird immer dann gesetzt (logisch 1), wenn das Ergebnis einer Operation 0 ist. Dies ist z. B. dann der Fall, wenn der Akkumulator nach Ausführung von *or b* oder *dec a* den Wert 00h aufweist. Fast alle arithmetisch/logischen Befehle beeinflussen das Zero-Flag (*Bild 1*).

Das Zero-Flag ist eine Grundvoraussetzung für Verzweigungen bzw. bedingte Sprünge in Maschinensprache: Bei *jp z, NN* führt der Prozessor nur dann einen Sprung nach NN aus, wenn das Zero-Flag gesetzt ist. Anderenfalls beachtet der Prozessor die Sprunganweisung nicht und wendet sich der nächsten Adresse zu.

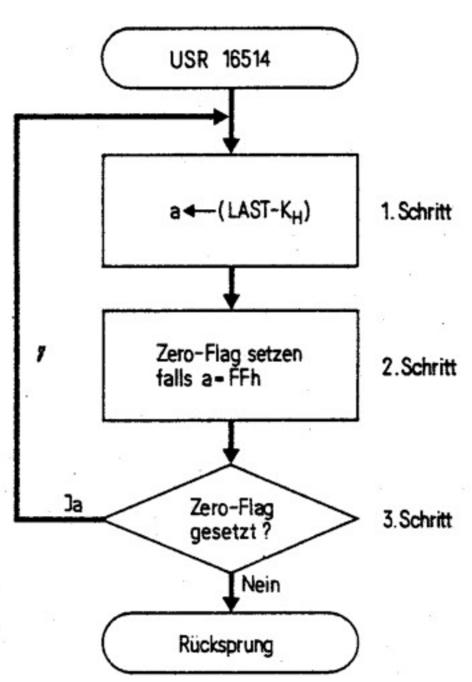
Im Gegensatz dazu führt die CPU bei *jp nz, NN* den Sprung nur dann aus, wenn das Zero-Flag nicht gesetzt ist. *Bild 2* zeigt alle Sprung-Befehle der Z-80-CPU, für deren Ausführung das Zero-Flag verantwortlich ist.

| add a, s<br>sub a, s | add a, N<br>sub a, N | and s<br>or s<br>xor s | and N<br>or N<br>xor N | inc s<br>dec s |
|---|---|---|---|---|
| s ≙ a, b, c, d, e, h, l oder (hl)<br>N ≙ 8-Bit-Zahl | | | | |

① **Flag-Beeinflussung:** Sobald das Ergebnis dieser arithmetisch/logischen Operationen 00 ist, wird das Zero-Flag gesetzt

| | | |
|---|---|---|
| jp z, NN | CA .. .. | jp NN wenn Zero-Flag gesetzt |
| jp nz, NN | C2 .. .. | jp NN wenn Z-Flag nicht gesetzt |
| call z, NN | CC .. .. | call NN wenn Z-Flag gesetzt |
| call nz, NN | C4 .. .. | call NN wenn Z-Flag nicht gesetzt |
| ret z | C8 | ret wenn Z-Flag gesetzt |
| ret nz | C0 | ret wenn Z-Flag nicht gesetzt |

② **Bedingte Sprünge:** Fast alle Operationen, bei denen der Programmzähler verändert wird, kann man vom Zustand des Zero-Flags abhängig machen




③ **Flußdiagramm zur Tastenabfrage:** Das Problem ist nach drei Schritten gelöst

### Verzweigungen in Maschinensprache

IF-THEN-Verzweigungen, wie wir sie von Basic her kennen, sind prinzipiell auch in Maschinensprache möglich. Versuchen wir z. B. ein Maschinenprogramm zu entwickeln, das solange wartet, bis eine beliebige Taste gedrückt wird, das also die Basic-Zeile ‚zz IF INKEY$ ="" THEN GOTO zz‘ ersetzt.

Das Programm muß in etwa der Idee des Flußdiagramms aus *Bild 3* entsprechen. Zuerst muß eines der beiden Bytes der Systemvariablen LAST-K in den Akkumulator gebracht werden (1. Schritt). Beide Bytes von LAST-K weisen den Wert FFh auf, wenn keine Taste gedrückt wird. Eben dann soll auch das Zero-Flag gesetzt werden (2. Schritt). Ist das Zero-Flag gesetzt, dann soll das Programm wieder von vorne ablaufen. Ansonsten erfolgt der Rücksprung ins Basic.

*Bild 4* zeigt die Lösung: Zuerst wird der Akkumulator mit dem Inhalt von LAST-K (höherwertiges Byte) geladen (4082h). Vom Akkumulator subtrahieren wir FFh (4085h). Falls vor dieser Operation FFh im Akkumulator stand, so ist jetzt dessen Inhalt 00 und das Zero-Flag ist gesetzt. Damit erfolgt ein

Sprung zum Anfang des Programms (4087h).

Ist dagegen eine Taste gedrückt, so wird das Zero-Flag nicht gesetzt. Der Sprung entfällt und die nächste Speicherzelle (408Ah) wird bearbeitet. Nach dem Aufruf des Programmes durch LET Q=USR 16514 erfolgt der Rücksprung ins Basic erst durch Drükken einer beliebigen Taste.

Einen kleinen Nachteil hat diese Methode, das Flag zu setzen, freilich doch: Da es hier mit einem arithmetischen Befehl geschieht, wird der Inhalt des Akkumulators durch *sub a, r* oder *sub a, N* ständig verändert.

Will man das vermeiden, so sollte auf den Compare-Befehl zurückgegriffen werden: *cp r* entspricht praktisch dem Befehl *sub a, r*, nur das das Ergebnis nicht in den Akkumulator geladen, sondern ausschließlich zum Setzen der Flags verwendet wird (*Bild 5*). Schreiben Sie zur Übung das Programm aus Bild 4 mit dem Befehl *cp FF*.

Programmier-Probleme werden häufig aus Geschwindigkeitsgründen in Maschinensprache gelöst: Maschinenprogramme sind an Tempo nicht zu überbieten! Manchmal ist dies aber zuviel des Guten und eine Art Bremse muß eingebaut werden.

## Verzögerungsschleife hilft bremsen

Solche Bremsen sind durchweg Verzögerungsschleifen. Dabei wird der Inhalt eines Register(-paars) in einer Schleife so oft um 1 verringert, bis der Inhalt 0 ist. Wie *Bild 6* zeigt, verwenden wir dafür z. B. das de-Registerpaar und laden es mit einer 16-Bit-Zahl, die für die Dauer der Verzögerung maßgebend ist. An diesem Punkt beginnt die Schleife (engl.: loop): Das de-Registerpaar wird dekrementiert. Dabei ist zu beachten, daß 16-Bit-Inkrementier- und Dekrementier-Befehle keine Wirkung auf Flags haben! Deshalb wird auch das Zero-Flag nicht gesetzt, wenn das de-Registerpaar 0 erreicht.

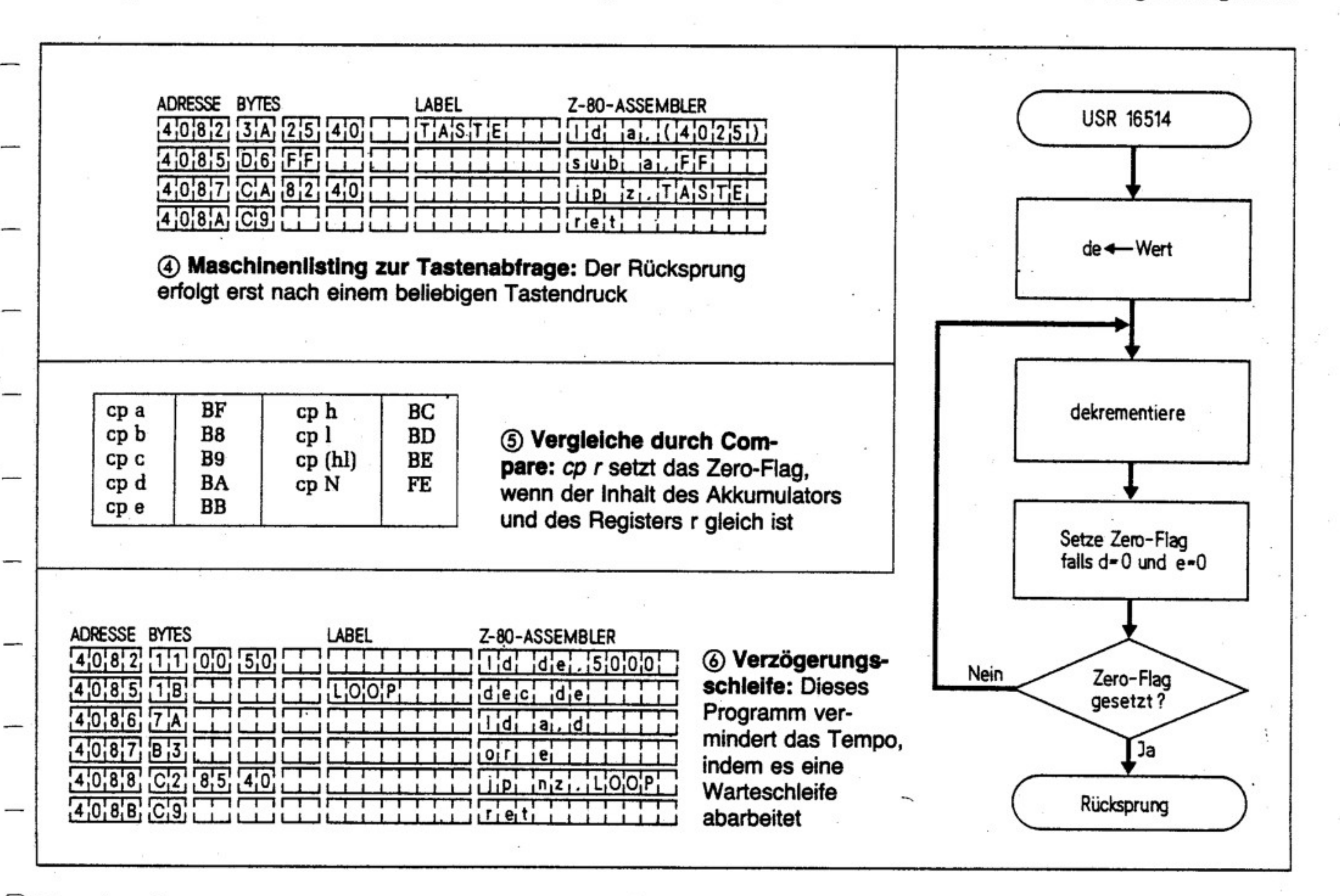
Wir müssen also einen kleinen Kunstgriff anwenden. Zuerst wird der Akkumulator mit dem d-Register geladen. Danach verknüpft *or e* den Akkumulator mit dem e-Register durch ein logisches ODER: Das Zero-Flag wird genau dann gesetzt, wenn der Akkumulator (mit der Kopie des d-Registers) und das e-Register den Wert 0 haben.

Solange aber das d-Register oder das e-Register ungleich 0 sind, wird das Zero-Flag nicht gesetzt, und es erfolgt ein Sprung zum Schleifenanfang. Nach Aufruf des Programmes durch LET Q=USR 16514 kehrt der Computer erst nach einiger Zeit ins Basic zurück. Finden Sie selbst heraus, welche Verzögerungszeiten sich erreichen lassen. Ein Tip: Die Systemvariable FRAMES ist dabei behilflich.

## Zero-Flag-Manipulation bei Einzelbit-Befehlen

In Teil 9 wurde gezeigt, wie durch bitweise Adressierung einzelne Bits eines Registers verändert werden. Nun ist der letzte Bereich der Einzelbit-Befehle an der Reihe:

Der Befehl *bit b, r* stellt fest, ob Bit b des Registers r gesetzt oder nicht gesetzt ist. Das Ergebnis wird dann wie üblich im Zero-Fag abgelegt. Generell gilt: Das Zero-Flag wird gesetzt, wenn das ausgewählte Bit b des Registers r den Signalzustand 0 aufweist, wenn es also zurückgesetzt ist. Im umgekehrten Fall wird das Zero-Flag nicht gesetzt.



| ADRESSE | BYTES | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|
| 4082 | 3A | 25 | 40 | TASTE | ld a,(4025) |
| 4085 | D6 | FF | | | sub a,FF |
| 4087 | CA | 82 | 40 | | jp z,TASTE |
| 408A | C9 | | | | ret |

④ **Maschinenlisting zur Tastenabfrage:** Der Rücksprung erfolgt erst nach einem beliebigen Tastendruck

| | | | |
|---|---|---|---|
| cp a | BF | cp h | BC |
| cp b | B8 | cp l | BD |
| cp c | B9 | cp (hl) | BE |
| cp d | BA | cp N | FE |
| cp e | BB | | |

⑤ **Vergleiche durch Compare:** *cp r* setzt das Zero-Flag, wenn der Inhalt des Akkumulators und des Registers r gleich ist

| ADRESSE | BYTES | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|
| 4082 | 11 | 00 | 50 | | ld de,5000 |
| 4085 | 1B | | | LOOP | dec de |
| 4086 | 7A | | | | ld a,d |
| 4087 | B3 | | | | or e |
| 4088 | C2 | 85 | 40 | | jp nz,LOOP |
| 408B | C9 | | | | ret |

⑥ **Verzögerungsschleife:** Dieses Programm vermindert das Tempo, indem es eine Warteschleife abarbeitet

Diese Einzelbit-Befehle sind in *Bild 7* aufgelistet.

Bisher wissen wir nur von der Existenz des Zero-Flags und wie man es nutzt. Es ist also an der Zeit zu erklären, wie das Flag computertechnisch realisiert ist.

## Flags stecken im f-Register

Alle Flags (auch die, die wir noch nicht kennen) sind im f-Register enthalten (*Bild 8*). Jedes Bit im f-Register stellt ein Flag dar: Wenn das Bit den Signalzustand 1 aufweist, ist das entsprechende Flag gesetzt; beim Signalzustand 0 ist das Flag nicht gesetzt. Bit 3 und Bit 5 des f-Registers sind unbenutzt.

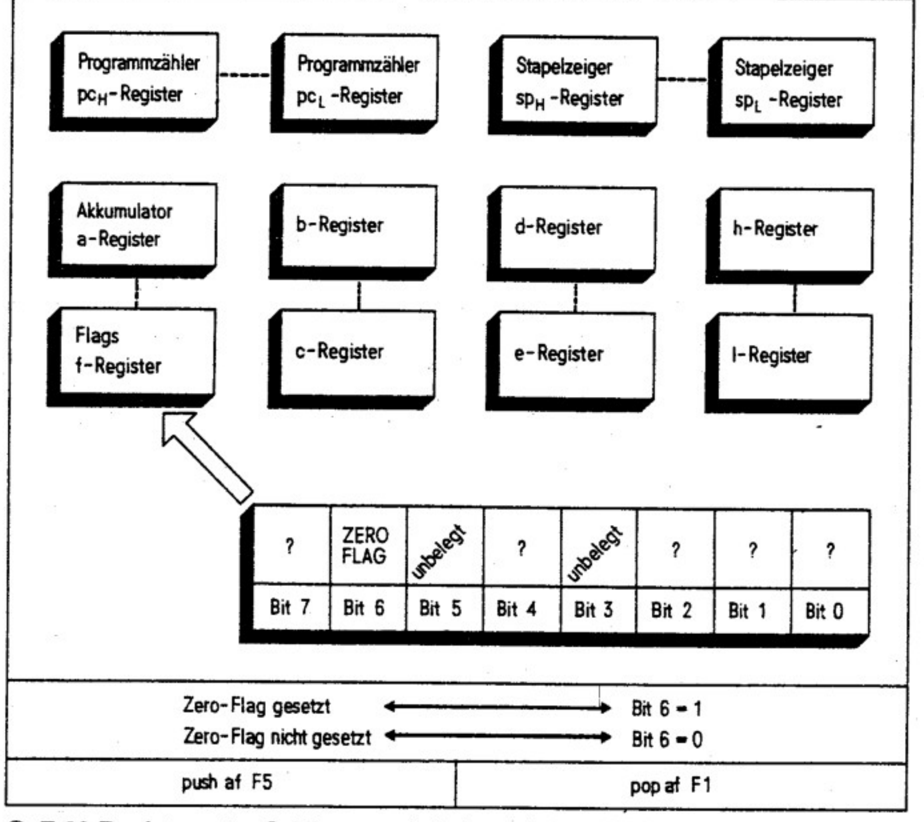
Der Akkumulator bildet mit dem f-Register das af-Registerpaar. Die Befehle *push af* und *pop af* erlauben sogar das Speichern dieses Registerpaares. Mit dem f-Register ist der Z-80-Registersatz im Rahmen dieser Serie fast vollständig. Die restlichen Flags werden in einem der nächsten Teile behandelt.

Klaus Herklotz

*(Wird fortgesetzt)*

| Register r | a | b | c | d | e | h | l | (hl) |
|---|---|---|---|---|---|---|---|---|
| bit 0, r | CB47 | CB40 | CB41 | CB42 | CB43 | CB44 | CB45 | CB46 |
| bit 1, r | CB4F | CB48 | CB49 | CB4A | CB4B | CB4C | CB4D | CB4E |
| bit 2, r | CB57 | CB50 | CB51 | CB52 | CB53 | CB54 | CB55 | CB56 |
| bit 3, r | CB5F | CB58 | CB59 | CB5A | CB5B | CB5C | CB5D | CB5E |
| bit 4, r | CB67 | CB60 | CB61 | CB62 | CB63 | CB64 | CB65 | CB66 |
| bit 5, r | CB6F | CB68 | CB69 | CB6A | CB6B | CB6C | CB6D | CB6E |
| bit 6, r | CB77 | CB70 | CB71 | CB72 | CB73 | CB74 | CB75 | CB76 |
| bit 7, r | CB7F | CB78 | CB79 | CB7A | CB7B | CB7C | CB7D | CB7E |

⑦ **Einzelbit-Befehle:** Bits werden auf ihren Wert hin kontrolliert. Das Zero-Flag signalisiert das Ergebnis




⑧ **Z-80-Registersatz:** Schlimmer wird's kaum! Jetzt fehlen nur noch die Index-Register. Das Zero-Flag ist Bit 6 des f-Registers

## ZX-81-Software:

# Maschinencode im Griff

Dieses Programm ermöglicht das Eingeben, Betrachten und Verändern von Maschinencode-Programmen. Zunächst fällt auf, daß das Programm hohe Zeilennummern, nämlich solche ab 9000, beansprucht. Das hat den Vorteil, daß man diesen Monitor in den Computer eingeben oder von Cassette laden und dann ein weiteres Programm (welches Maschinensprache-Routinen verwenden soll) wie gewohnt mit niedrigen Zeilennummern eintippen kann. Irgendwo im Programm wird Speicherplatz für den Maschinencode reserviert, üblicherweise durch ein REM-Statement mit entsprechend vielen Bytes hinter REM.

Durch den Befehl RUN 9000 wird der Monitor aufgerufen, der sofort nach der Anfangsadresse des Maschinencodes fragt. Diese ist dezimal einzugeben (beispielsweise 16514). Nun wird hexadezimal der Inhalt der ersten zwölf Bytes ab dieser Adresse angezeigt. Drückt man NEWLINE, so können die nächsten Speicherplätze betrachtet werden. Falls man die gerade angezeigten Bytes verändern möchte, ist einfach der gewünschte neue Inhalt einzutippen. Dies dürfen ein bis zwölf Bytes sein; Die Veränderung findet ab der angezeigten Adresse in der erforderlichen Länge statt. Fehlerhafte Eingaben (ungerade Stellenzahl oder für Hex-Code unzulässiges Zeichen) werden ignoriert. Die Eingabe eines „S" (für STOP) bricht das Programm ab; ein „A" bewirkt, daß eine neue Startadresse erfragt wird.

Michael Schramm

```
9000 PRINT AT 19,14;"MONI"
9010 PRINT AT 21,0;"STARTADRESSE
:"
9020 INPUT A
9030 LET A=INT ABS A
9040 SCROLL
9050 PRINT A;TAB 8;
9060 FOR I=0 TO 11
9070 LET M=INT (PEEK (A+I)/16)
9080 PRINT CHR$ (M+28);CHR$ (PEE
K (A+I)-16*M+28);
9090 NEXT I
9100 INPUT D$
9110 IF D$="S" THEN STOP
9120 IF D$="A" THEN GOTO 9010
9130 IF NOT LEN D$ THEN GOTO 924
0
9140 IF LEN D$<>2*INT (LEN D$/2)
 THEN GOTO 9100
9150 FOR I=1 TO LEN D$
9160 IF D$(I)<"0" OR D$(I)>"F" T
HEN GOTO 9100
9170 NEXT I
9180 PRINT AT 21,8;(D$+"
                   ")( TO 24)
9190 FOR I=1 TO LEN D$ STEP 2
9200 POKE A,16*CODE D$(I)+CODE D
$(I+1)-476
9210 LET A=A+1
9220 NEXT I
9230 GOTO 9040
9240 LET A=A+12
9250 GOTO 9040
```

**Hex-Monitor:** Er hilft beim Schreiben von Maschinencode-Programmen

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

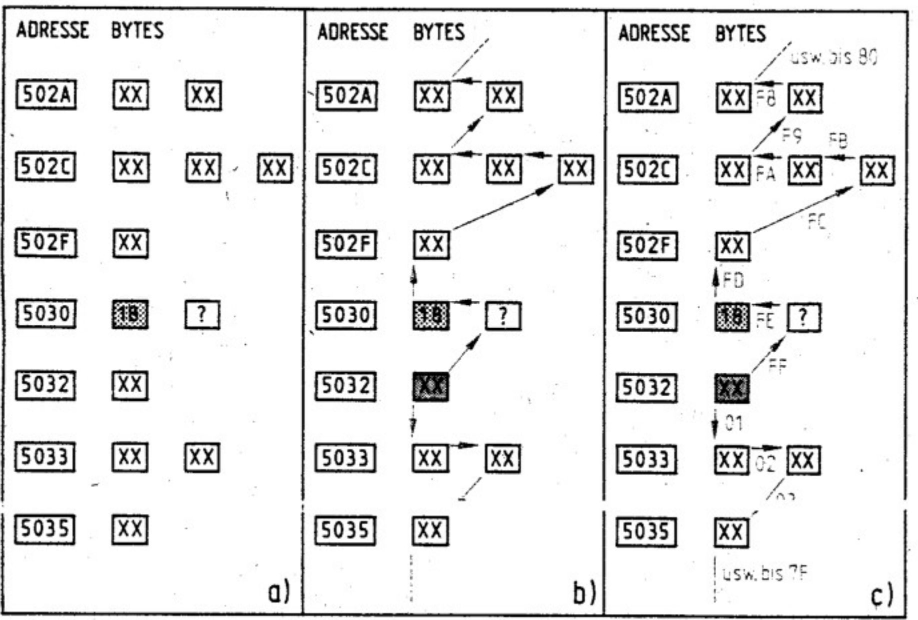
## Teil 12: Der relative Sprung

**Der Z-80-Mikroprozessor erlaubt zwei Sprungarten: Die relative, die wir hier behandeln werden, und die absolute, die wir bereits kennen.**

Bei absoluten Sprüngen wird dem Prozessor die Zieladresse durch zwei Adreß-Bytes ohne viel Geplänkel mitgeteilt. Einen gewaltigen Nachteil hat die Angelegenheit aber doch: Wenn ein Maschinenprogramm verschoben wird – etwa durch ein nachträglich eingefügtes Byte zur Behebung eines Programmierfehlers – dann müssen alle betroffenen Adreß-Bytes nachgestellt werden. Grund genug also, die relative Sprungart anzuwenden.

| | | |
|---|---|---|
| jr E | 18 | Relativer Sprung um die Entfernung E<br>Anschaulich: jr E ≙ jp pc ± E |
| jr z, E | 28 | jr E, wenn Zero-Flag gesetzt |
| jr nz, E | 20 | jr E, wenn Zero-Flag nicht gesetzt |

① **Relative Sprünge:** Man spart wertvollen Speicherplatz und kann Programmteile verschieben




② **Sprungzielberechnung:** Ein beliebiges Maschinenprogramm (a) hat z. B. unter der Adresse 5030h einen relativen Sprungbefehl. Ausgangspunkt für die Berechnung der Sprungweite ist die Adresse 5032h, weil der Programmzähler vor dem Sprung auf diese Adresse zeigt (b). Bei Vorwärtssprüngen zählt man aufwärts von 01 an, bei Rückwärtssprüngen abwärts von FF an (c)

### Relative Sprünge erfordern Geschick

Bei relativen Sprüngen benötigt der Prozessor an Stelle der Zieladresse die relative Sprungweite. Der Programmierer muß sich überlegen, um wie viele Speicherzellen der Sprung vorwärts oder rückwärts erfolgen soll.

Die relativen Sprungbefehle und ihre Codes zeigt *Bild 1*. Dem Operationscode folgt nicht die Zieladresse, sondern die Sprungweite, die durch ein einziges Byte angegeben wird.

Mit relativen Sprüngen spart man deshalb gegenüber absoluten Sprüngen immer ein Byte! Weiterhin können Programmteile verschoben werden, ohne daß Korrekturen notwendig sind: Relative Sprungweiten sind unabhängig von Adressen und demnach überall gültig. Wie aber wird die Sprungweite angegeben?

Das Maschinenprogramm in *Bild 2a* soll ein x-beliebiges sein. Lediglich Adresse 5030h enthält den Operationscode des relativen Sprungs. Unter der Adresse 5031h ist dann die verschlüsselte Sprungweite einzugeben. Dazu ist die nächste Speicherzelle 5032h markiert (Bild 2b). Sie soll Ausgangspunkt aller folgenden Überlegungen sein, denn auf sie zeigt der Programmzähler nach der Befehlsausführung.

Von Adresse 5032h aus sind alle zu überspringenden Bytes durch Pfeile verbunden und durchnumeriert (Bild 2c): Vorwärts-Sprünge zum Ende des Speichers hin in steigender Reihenfolge ab 01h; Rückwärts-Sprünge zum Anfang des Speichers hin in fallender Reihenfolge ab FFh.

Diese Hex-Zahlen an den Pfeilen geben den Wert an, der jeweils in Adresse 5031h eingesetzt werden muß, um von dort zu der Adresse zu springen, auf die der entsprechende Pfeil deutet: Soll z. B. ein Rückwärts-Sprung zur Adresse 502Ch programmiert werden, so muß Speicherzelle 5031h den Wert FAh erhalten. Bei einem Sprung nach 5035h müßte 03h eingesetzt werden. Daran läßt sich erkennen, daß die Sprungweite bei relativen Sprüngen begrenzt ist.

Liegt der Wert der Sprungweite zwischen 01h und 7Fh, so erfolgt ein Sprung nach vorne. Liegt der Wert dagegen zwischen 80h und FFh, so wird eine rückwärtige Zieladresse angepeilt.

| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | 3A | 25 | 40 | | TASTE | ld a,(4025) |
| 4085 | FE | FF | | | | cp FF |
| 4087 | 28 | F9 | | | | jr z TASTE |
| 4089 | C9 | | | | | ret |

③ **Tastendruck:** Ein altes Programm in neuer Auflage wartet bis eine Taste gedrückt wird

## Das Tastendruck-Problem wird neu gelöst

In Teil 11 wurde eine Maschinen-Routine behandelt, die so lange wartet, bis eine Taste gedrückt wird. Der darin verwendete absolute Sprung soll nun durch einen relativen ersetzt werden. Die einzige Schwierigkeit dürfte in der Angabe der Sprungweite liegen. Man betrachte deshalb das Listing aus *Bild 3* und stelle sich wieder die numerierten Pfeile vor! Es wird dann bestimmt klar, daß F9h als Sprungweite gerechtfertigt ist.

Wie auch schon im letzten Teil, erfolgt nach dem Aufruf des Programms mit LET Q=USR 16 514 der Rücksprung ins Basic erst durch Drücken einer beliebigen Taste.

## So berechnet der Computer die tatsächliche Sprungweite

Soll der Z-80-Mikroprozessor einen relativen Sprung ausführen, so muß er zuerst die Sprungrichtung ermitteln. Die Grenze zwischen den beiden Richtungen liegt wie gezeigt bei 7Fh bzw. 80h. Betrachten wir beide Zahlen im binären Zahlensystem:

7Fh = 0111 1111b und
80h = 1000 0000b

Die Richtung des Sprunges kann praktisch von einem Bit abhängig gemacht werden: Ist Bit 7 nicht gesetzt, dann erfolgt der Sprung nach vorne. Sollte dagegen Bit 7 gesetzt sein, so kündigt das einen Rückwärts-Sprung an.

Bei Vorwärts-Sprüngen ist noch alles klar: Als Sprungweite wird einfach die Anzahl der zu überspringenden Bytes angegeben. Bei Rückwärts-Sprüngen beginnen aber die Schwierigkeiten. Denn wie errechnet der Prozessor aus dem verschlüsselten Wert der Sprungweite die Anzahl der zu überspringenden Bytes?

Des Rätsels Lösung lautet „Zweier-Komplement“. Sollte Bit 7 der verschlüsselten Sprungweite gesetzt sein, dann dreht der Prozessor alle Bits um (Fachsprache: Er invertiert sie oder bildet das Komplement) und zählt 1 dazu. Das Ergebnis liefert die tatsächliche Anzahl der zu überspringenden Bytes. Dazu ein Beispiel mit bekannten Werten:

Sprungweite: FAh = 1111 1010b
Bits invertiert: 0000 0101b
1 dazu: (06h) = 0000 0110b

Sollte die verschlüsselte Sprungweite FAh betragen, so muß der Prozessor also 06h Speicherzellen nach hinten „überspringen“. Man überzeuge sich von der Richtigkeit des Ergebnisses durch Nachzählen der Pfeile in Bild 2c bis zur Zelle FAh. Klaus Herklotz

*(Wird fortgesetzt)*

ZX-81-Hardwaretip:

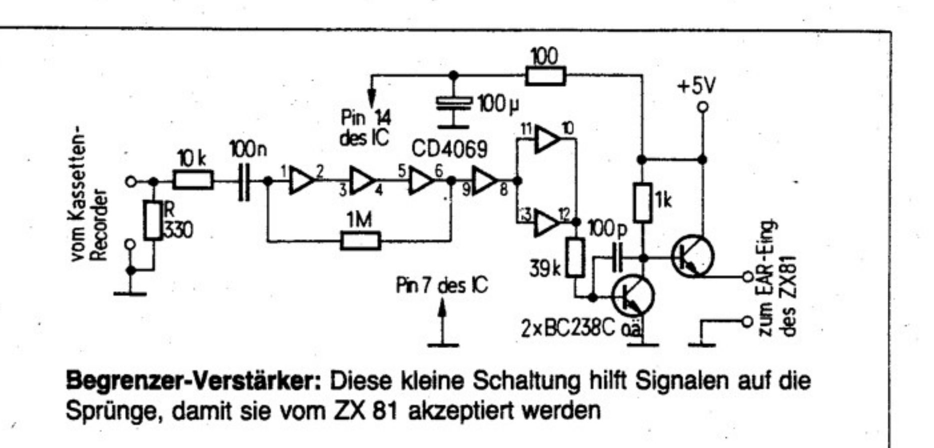
# Signalverbesserung bei LOAD

Das Laden von Programmen, die auf einem anderen Recorder aufgezeichnet worden sind, bereitet oft Probleme. Das liegt meist an einer abweichenden Tonkopf-Einstellung, die einen zu geringen Pegel des wiedergegebenen Signals zur Folge hat. Der Einsatz der hier angegebenen Schaltung führt in derartigen Situationen oft doch noch zum Erfolg. Freilich darf man keine Wunder erwarten; wenn das Signal zu schwach ist, so hilft höchstens die Verwendung eines anderen Kassettenrecorders oder das Verstellen des Tonkopfes.

Die Funktion der Schaltung ist recht einfach: Das Programm-Signal gelangt auf den Eingang eines Sinus-zu-Rechteck-Formers, der mit dem CMOS-IC CD 4069 realisiert ist (*Bild*). Das Rechteck-Signal erfährt durch die beiden Transistoren noch eine kräftige Stromverstärkung, um den niederohmigen Eingang EAR des ZX 81 ansteuern zu können. Der 100-pF-Kondensator blokkiert eingestreute Hochfrequenz und verhindert Eigenschwingungen der Schaltung. Da die Schaltung nur wenig Strom aufnimmt, kann sie ohne weiteres vom 5-V-Spannungsregler des ZX 81 mitversorgt werden.

Anstelle des Lautsprecher/Ohrhörer-Ausgangs kann jetzt auch der DIN-Anschluß eines Recorders das Signal für den ZX 81 liefern; der Wert des Widerstands R muß in diesem Fall entsprechend vergrößert werden (etwa 47 kΩ).

Michael Schramm




**Begrenzer-Verstärker:** Diese kleine Schaltung hilft Signalen auf die Sprünge, damit sie vom ZX 81 akzeptiert werden

Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 13: Nachbilden von FOR-NEXT-Schleifen

**Den einfachen relativen Sprung haben wir im vorangegangenen Teil kennengelernt. Mit einem speziellen Sprungbefehl sind auch FOR-NEXT-Schleifen möglich – nicht so komfortabel wie in Basic – aber immerhin...**

In Teil 12 wurde gezeigt, daß die Z-80-CPU zur Berechnung der Sprungweite eines relativen Sprungs den Inhalt des Akkumulators komplementieren muß. Im Z-80-Befehlsvorrat gibt es dafür zwei arithmetisch/logische Befehle: Der Befehl *cpl* mit dem Code 2F invertiert alle Bits im Akkumulator, bildet also das sogenannte Einer-Komplement. Der Befehl *neg* mit dem Zwei-Byte-Code ED44 invertiert zuerst alle Bits im Akkumulator und zählt dann 1 dazu, was dem Zweier-Komplement gleichkommt (siehe Teil 12). Erproben Sie die beiden Befehle selbständig!

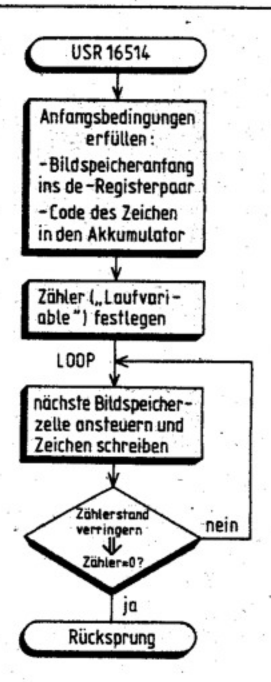
### Leistungsstarker Befehl zur Schleifenbildung

Schleifen sind in Basic ein wesentliches Element zur wiederholten Ausführung von Programmteilen. Zur einfachen Schreibweise solcher Wiederholungen bietet Basic dafür die Befehle FOR und NEXT.

In der Z-80-Maschinensprache sind FOR-NEXT-Schleifen ebenfalls realisierbar, jedoch nicht so komfortabel wie in Basic. Als „Laufvariable" kommt nur das b-Register in Frage und die Schrittweite beträgt stets -1.

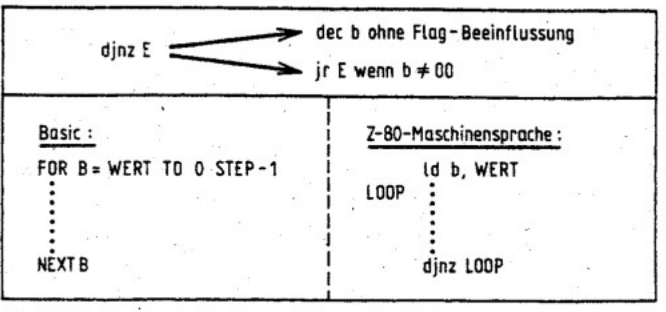
Zu Beginn jeder Schleife erhält das b-Register einen Wert mit der Anzahl der Durchläufe zugewiesen. Am Schleifenende erscheint dann der Zwei-Byte-Befehl *djnz E* (Abkürzung: decrement and jump if not zero) mit dem Operationscode 10. Er erfüllt gleich zwei Funktionen auf einmal: Zum einen wird der Inhalt des b-Registers um 1 vermindert, ohne dabei Flags zu beeinflussen, zum anderen erfolgt ein relativer Sprung um die Sprungweite E, wenn das b-Register noch nicht auf 0 ist *(Bild 1)*. Sollte es 0 sein – die Schleife ist dann beendet – wird ganz einfach die nächste Speicherzelle bearbeitet.

Die Sprungweite E ist, wie bei relativen Sprüngen üblich, in einem Byte verschlüsselt, das dem Operationscode




| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | ED | 5B | 0C | 40 | | ld de,(400C) |
| 4086 | 3E | 26 | | | | ld a,26 |
| 4088 | 06 | 06 | | | | ld b,06 |
| 408A | 13 | | | | LOOP | inc de |
| 408B | 12 | | | | | ld (de),a |
| 408C | 10 | FC | | | | djnz LOOP |
| 408E | C9 | | | | | ret |

② **Schleife in Maschinensprache:** Dieses Programm schreibt die Zeichenfolge „AAAAAA" auf den Bildschirm. Die Zahl der A's wird vom Inhalt des b-Registers bestimmt (Adresse 4088h)




① **Schleifenbefehl:** Der 2-Byte-Befehl djnz E vereint zwei andere Z-80-Befehle. Damit lassen sich FOR-NEXT-Schleifen fast wie in Basic programmieren

folgt. Meistens liegt die Zieladresse direkt hinter dem Befehl *ld b, WERT* (siehe Bild 1) am Anfang der Schleife (LOOP).

## Die Übersicht behalten mit dem djnz-Befehl

Der djnz-Befehl vereint zwei bekannte Z-80-Befehle. Somit spart man bei seiner Verwendung immer ein Byte, und – das ist noch viel entscheidender – man gewinnt ein hohes Maß an Übersichtlichkeit! Ein Anwendungsbeispiel verdeutlicht das:

Es soll eine Maschinenroutine entworfen werden, die eine vorgegebene Anzahl gleicher Buchstaben (z. B. „AAAA") auf den Bildschirm schreibt. Betrachten wir dazu gleich das Flußdiagramm und Maschinenlisting (*Bild 2*).

Zur Abwechslung erhält nicht das hl-, sondern das de-Registerpaar die Adressen der angesteuerten Bildspeicherzellen zugewiesen: Die Systemvariable D-FILE liefert dazu die Adresse der Bildspeichergrenze (Adresse 4082h im Maschinenlisting). Dann wird der Akkumulator mit dem Hex-Code des zu druckenden Zeichens geladen (4086h) und die Anzahl der Schleifendurchläufe ins b-Register gebracht (4088h). Danach wird der Schleifenbeginn in der nächsten Speicherzelle mit LOOP etikettiert.

Im Laufe der Schleife selbst, wird der Inhalt des de-Registerpaars erhöht (408Ah) und der Code des gewünschten Zeichens in die durch das de-Registerpaar adressierte Speicherzelle geladen (408Bh). Den Abschluß der Schleife bildet der djnz-Befehl mit der Sprungweitenangabe (408Ch).

Klaus Herklotz

*(Wird fortgesetzt)*

ZX-81-Software:

# Malen am Bildschirm

Das nachfolgend beschriebene kurze Programm (*Bild 1*) ermöglicht beliebige Grafiken am Bildschirm. Damit lassen sich z. B. grafisch gestaltete Geburtstagsgrüße, Grundrisse, Irrgärten oder einfach der Phantasie entsprungene Figuren zeichnen. Die „Strichstärke" ist durch den PLOT-Befehl gegeben (*Bild 2*).

Die Zeilen 20 bis 50 bestimmen den Ausgangszustand (Daten für: Zeichenpunkt blinkend in der linken unteren Ecke des Bildschirms); die Zeilen 50 bis 140 werden ständig durchlaufen. Sie steuern den Zeichenpunkt, bewirken die Ausgabe am Drucker usw. in Abhängigkeit davon, welche Taste gedrückt ist.

Besonders interessant sind die Zeilen 120 und 130, denn hier wird mit logischen Aussagen gerechnet. Viele Basic-Programmierer wissen gar nicht, daß es diese äußerst nützliche Möglichkeit überhaupt gibt. Es werden die X- und Y-Koordinaten des nächsten Punktes in Abhängigkeit von den alten X- und Y-Werten und von A$ (zuletzt gedrückte Taste) bestimmt, wobei die zulässigen Ober- und Untergrenzen für X und Y zu berücksichtigen sind. Selbstverständlich ist das auch durch einige IF–THEN-Anweisungen machbar; kürzer und im Programmlauf schneller geht's auf die hier gewählte Weise (siehe auch Kapitel 10 des ZX-81-Handbuchs).

Der ZX 81 benutzt die Zahlen 0 und 1 für die Ergebnisse FALSCH und WAHR von logischen Aussagen. Die Aussage (A$ = „8" AND X < 63) nimmt also den Wert 1 an, falls die Taste 8 (Cursor) gedrückt und X kleiner als 63 ist. Somit darf X weiter erhöht werden, wenn die rechte Bildkante noch nicht erreicht ist. Entsprechendes gilt für die anderen logischen Aussagen, wobei zu beachten ist, daß der ZX 81 nicht nur 1, sondern jede von 0 abweichende Zahl als logischen Wert WAHR anerkennt; daher kann für X <> 0 einfach X geschrieben werden.

Gestartet wird das Programm durch RUN. Es erscheint sofort in der linken unteren Ecke blinkend der Zeichenpunkt. Die folgenden Tasten sind mit Funktionen belegt.

5: Zeichenpunkt nach links
6: Zeichenpunkt nach unten
7: Zeichenpunkt nach oben
8: Zeichenpunkt nach rechts
0: Lösch-Betriebsart (Zeichenpunkt blinkt)
1: Zeichnen-Betriebsart
N: Zeichenpunkt zum Ausgangspunkt
S: SAVE (Programm und Bildschirminhalt)
C: COPY (Grafik auf Drucker geben)
BREAK unterbricht das Programm.

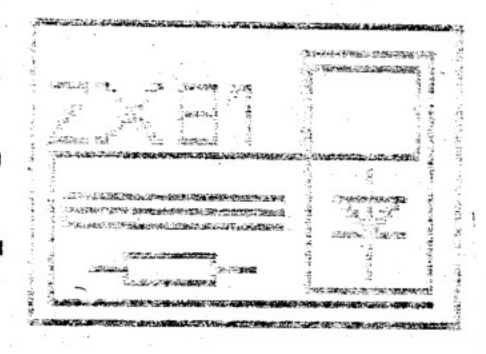
**Variablenbelegung:**

X: X-Koordinate des aktuellen Punktes (0 bis 63)
Y: Y-Koordinate des aktuellen Punktes (0 bis 43)
M$: Betriebsart
A$: zuletzt gedrückte Taste

Michael Schramm

```
  10 REM GRAPHIK
  20 LET X=0
  30 LET M$="0"
  40 LET Y=X
  50 PLOT X,Y
  60 LET A$=INKEY$
  70 IF A$="C" THEN COPY
  80 IF A$="S" THEN SAVE "GRAPHI
█"
  90 IF A$="0" OR A$="1" THEN LE
T M$=A$
 100 IF M$="0" THEN UNPLOT X,Y
 110 IF A$="N" THEN RUN
 120 LET X=X+(A$="8" AND X<63)-(
A$="5" AND X)
 130 LET Y=Y+(A$="7" AND Y<43)-(
A$="6" AND Y)
 140 GOTO 50
```

**① Programmlisting „Grafik":** Die Bewegung des Zeichenpunktes fußt auf logischen Aussagen



**② Grafikbeispiel:** Etwa 10 min waren erforderlich, um dieses Bild mit Hilfe der Cursortasten zu zeichnen

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 14: Übung macht den Meister

**Jetzt geht es darum, die bisher erworbenen Kenntnisse in der Z-80-Programmierung systematisch zum Bewältigen einer Übungsaufgabe einzusetzen.**

Vorbereitungen treffen
KUGEL
Rückspung vorbereiten
Hauptprogramm
Unterprogramm

① **Blockdiagramm:** Das ist der erste Schritt des Programmentwurfs zum Kugel-Problem. Dieses Diagramm zeigt zugleich den Ablauf des Hauptprogramms

Diesmal gilt es, selbst eine Aufgabe zu lösen. Schreiben Sie eine Maschinenroutine, die eine symbolisierte Kugel am Bildschirm waagerecht hin- und herbewegt. Sobald eine Taste gedrückt wird, soll der Rücksprung zum Basic-Programm den Vorgang unterbrechen, wobei das bc-Registerpaar zugleich die Bildspeicheradresse der letzten Kugel-Position mitschleppt.

Damit es über die Aufgabe keine Mißverständnisse gibt, zäumen wir das Pferd von hinten auf: Geben Sie zunächst mit einem Eingabeprogramm folgende 57 Bytes ein:

```
2A 0E 40 CD 8B 40 4D 44 C9 06 0F 36
00 23 36 34 3A 25 40 FE FF C0 CD B2
40 10 F0 06 0F 36 00 2B 36 34 3A 25 40
FE FF C0 CD B2 40 10 F0 C3 8B 40 11
00 03 1B 7A B3 20 FB C9
```

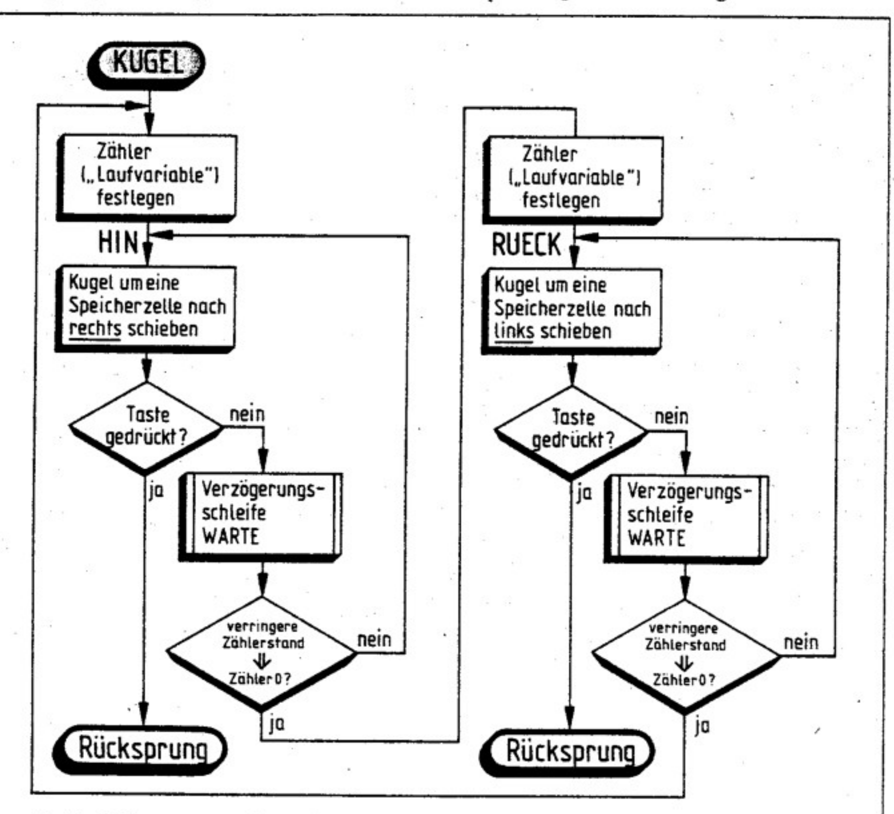
Rufen Sie das Programm nun mit PRINT AT 9,9;USR 16514 auf. Die gewünschte Bewegung wird sichtbar, wobei ein Tastendruck den Rücksprung einleitet. Erst mit diesem Ziel vor Augen, können wir das Listing entwickeln. Die Umsetzung der Idee vollzieht sich dabei in vier Schritten:

### 1. Schritt: Blockdiagramm

Der erste Grobentwurf ist grundsätzlich ein Blockdiagramm (*Bild 1*). Für das Kugel-Problem reichen zwei Nebenblöcke und ein Hauptblock völlig aus. Aufgabe des Nebenblocks „Vorbereitungen treffen" ist es, mit Hilfe der Systemvariablen DF-CC (momentane PRINT-Position), lediglich die Adresse der benötigten Bildspeicherzelle in ein Registerpaar zu laden.

Ein solcher Block ist in fast jedem längeren Maschinenprogramm zu finden. Mit ihm werden hauptsächlich Systemvariablen abgefragt bzw. Register und Registerpaare mit Startwerten versehen.

Der Hauptblock „KUGEL" ist für die Bewegung der Kugel bis hin zum Tastendruck verantwortlich. Vom letzten Block wird schließlich der Rücksprung in das Basic-Programm vorbereitet: Das bc-Registerpaar erhält dazu die Adresse der zuletzt angepeilten Bildspeicherzelle. Der Block „Rücksprung vorbereiten" ist in Maschinenprogrammen nur dann sinnvoll, wenn man das bc-Registerpaar auch im zugehörigen Basic-Programm benötigt.




② **Flußdiagramm:** Das Unterprogramm „KUGEL" ist der wichtigste Teil des Hauptprogramms. Das Unterprogramm selbst ruft mit der Verzögerungsschleife „WARTE" ein weiteres Unterprogramm auf

## 2. Schritt: Feinentwurf

Im Feinentwurf gilt es für jeden größeren Block ein Flußdiagramm auszuarbeiten. Beim Kugel-Problem genügt ein Diagramm für den Hauptblock. Wie *Bild 2* zeigt, besteht das Programm „KUGEL" aus zwei Abschnitten: Im ersten wird die symbolisierte Kugel nach rechts bewegt (HIN), im zweiten Abschnitt nach links (RUECK). Am Ausgangspunkt angelangt, wiederholt sich dieser Zyklus.

Sowohl die Hin- als auch die Rück-Bewegung der Kugel vollzieht sich in einer Schleife: Zunächst wird die Kugel in die gerade gültige Bewegungsrichtung verschoben, das heißt, der Code der Kugel wird in die nächsthöhere oder nächstniedrigere Speicherzelle geladen, danach das Tastenfeld abgefragt und eventuell ins Hauptprogramm zurückgesprungen. Sollte keine Taste gedrückt sein, wird die Verzögerungsschleife „WARTE" aufgerufen.

Die Hauptschleife bleibt solange aktiv, bis die Kugel am Ziel angelangt ist. Danach wird die umgekehrte Bewegungsrichtung eingeschlagen.

## 3. Schritt: Registervergabe

Nachdem nun der Programmablauf feststeht, kann mit der Vergabe der Register begonnen werden. Im Gegensatz zur Variablenverteilung in Basic muß dies mit allergrößter Sorgfalt geschehen! Dabei hat es sich bestens bewährt, das hl-Registerpaar mit den schwersten 16-Bit-Aufgaben zu belegen: Beim Kugel-Programm erhält es die Bildspeicheradresse für die Position der Kugel. Außerdem wird das de-Registerpaar als Zähler der Verzögerungsschleife verwendet.

Das b-Register findet als 8-Bit-Zähler der Hauptschleife Verwendung, während wir dem Akkumulator kurzfristige Aufgaben (Datenträger beim Flagsetzen) zuteilen. Geradezu unbedeutend ist der Teil des Programmes, in dem das bc-Registerpaar zwecks Rücksprung eine Kopie des hl-Registerpaares erhält.

## 4. Schritt: Listing

Erst mit dem letzten Schritt beginnt die eigentliche Programmausarbeitung; Labels werden dabei besonders nutzbringend eingesetzt. Wer selbständig ein Maschinen-Listing erstellen will, sollte sich dann an folgendes Patentrezept halten:

○ Man teile ein Blatt Papier in vier Spalten auf und beschrifte diese mit „Adresse", „Bytes", „Label" und „Assembler".

○ Zuerst die hinteren beiden Spalten ausfüllen! Dabei entsteht ein Assembler-Listing, das völlig unabhängig von Adressen ist (*Bild 3a*). 8-Bit-Datenbytes können dabei durch Labels ersetzt werden (z. B. ZÄHLER).

○ Erst jetzt mit den Hex-Codes beginnen! Dazu alle Bytes (bis auf die Adressen-Bytes) in die entsprechende Spalte eintragen. Adressen-Bytes z. B. durch Kästchen kennzeichnen (*Bild 3b*).

○ Danach die Adressen selbst eintragen (*Bild 3c*).

○ Abschließend die relativen Sprungweiten mit der Pfeilmethode (siehe Teil 12) ausarbeiten und die Zieladressen der absoluten Sprünge eintragen (*Bild 3d*).

Nun ist es an der Zeit, einen Lösungsvorschlag zum Kugel-Problem zu geben. Verfolgen Sie ihn aufmerksam, denn es werden einige Aufgaben folgen (*Bild 4*).

Beginnen wir mit dem Blockdiagramm aus Bild 1; es ergibt im Z-80-Listing die Adressen 4082h bis 408Ah: Das hl-Registerpaar erhält die augenblickliche PRINT-Position im Bildspeicher zugewiesen (4082h), das Unterprogramm „KUGEL" wird aufgerufen (4085h) und anschließend das hl- ins bc-Registerpaar kopiert. Den Abschluß des gesamten Programms bildet der Rücksprung in Zeile 408Ah.

Danach ist die erste Unterprogrammebene mit dem Unterprogramm „KUGEL" (408Bh bis 40B1h) an der Reihe. Dort ist festgelegt, daß die Kugel in jede Richtung 16 Speicherzellen weit rollen soll: Zählregister b erhält also die Anzahl der Schleifendurchläufe (408Bh). In der Schleife selbst wird die zuletzt geschriebene Kugel gelöscht (408Dh), die nächste Speicherzelle angepeilt (408Fh) und der Code der Kugel (34h) dort eingeschrieben (4090h).

Das Tastenfeld wird anschließend abgefragt (4092h) und dementsprechend das Zero-Flag gesetzt (4095h). Sollte eine Taste gedrückt sein, so er-

a)

| Adresse | Bytes | Label | Assembler |
|---|---|---|---|
| | | | ld hl , (DF-CC) |
| | | | call KUGEL |
| | | | ld c , l |
| | | | ld b , h |
| | | | ret |
| | | KUGEL | ld b , ZÄHLER |

b)

| Adresse | Bytes | Label | Assembler |
|---|---|---|---|
| | 2A 0E 40 | | ld hl , (DF-CC) |
| | CD □□ | | call KUGEL |
| | 4D | | ld c , l |
| | 44 | | ld b , h |
| | C9 | | ret |
| | 06 0F | KUGEL | ld b , ZÄHLER |

c)

| Adresse | Bytes | Label | Assembler |
|---|---|---|---|
| 4082 | 2A 0E 40 | | ld hl , (DF-CC) |
| 85 | CD □□ | | call KUGEL |
| 88 | 4D | | ld c , l |
| 89 | 44 | | ld b , h |
| 8A | C9 | | ret |
| 8B | 06 0F | KUGEL | ld b , ZÄHLER |

d)

| Adresse | Bytes | Label | Assembler |
|---|---|---|---|
| 4082 | 2A 0E 40 | | ld hl , (DF-CC) |
| 85 | CD [8B] [40] | | call KUGEL |
| 88 | 4D | | ld c , l |
| 89 | 44 | | ld b , h |
| 8A | C9 | | ret |
| 8B | 06 0F | KUGEL | ld b , ZÄHLER |

③ **So entsteht ein Maschinenprogramm:** Zuerst das Listing auf Labels beschränken (a), anschließend so weit wie möglich die Hex-Codes ermitteln (b), dann die Adressen ergänzen (c) und letztlich die Sprungadressen eintragen (d)

folgt ein Rücksprung ins Hauptprogramm (4097h), ansonsten wird die Verzögerungsschleife „WARTE“ aufgerufen (4098h). Abschließend erfolgt ein Sprung zum Anfang der Schleife, solange das b-Register durch den djnz-Befehl noch nicht zu 00h geworden ist (409Bh).

Der nun folgende Teil (409Dh bis 40AEh) ist fast gleich mit dem eben behandelten (6408Bh bis 409Ch). Lediglich bei Adresse 40A1h wird die umgekehrte Bewegungsrichtung eingeschlagen. Der absolute Sprung bei Adresse 40AFh sorgt dafür, daß der gesamte Bewegungsvorgang wiederholt wird.

Die Verzögerungsschleife „WARTE“ (40B2h bis 40BAh) beansprucht die zweite und letzte Unterprogrammebene. Das hohe Bewegungstempo kommt durch den geringen Startwert 300h des Zählregisters de zustande. Wem die Bewegung zu langsam ist, der muß lediglich den Startwert vermindern.

## Freiwillige Selbstkontrolle

Lösen Sie die folgenden Aufgaben erst, wenn Sie das Listing (Bild 4) nachvollzogen haben. Halten Sie unbedingt die Reihenfolge der Aufgaben ein!

○ Ersetzen Sie den absoluten Sprung bei Adresse 40AFh durch einen relativen Sprung. Kontrolle: Das System darf nicht zusammenbrechen. Hilfe: Teil 12 dieser Serie.

○ Ersetzen Sie die beiden djnz-Befehle durch Dekrementierbefehle und relative Sprünge. Kontrolle: Auch hier darf das System nicht zusammenbrechen. Hilfen: Teil 11 und Teil 12. Hinweis: Die Sprungweite ändert sich!

○ Vertauschen Sie nun das hl- mit dem bc-Registerpaar: Das h-Register soll Zähler der Schleife sein, wogegen das bc-Registerpaar die Position der Kugel erhält. Einen Lösungsvorschlag finden Sie im nächsten Teil. Hilfe: Erstellen Sie das Listing nach dem zuvor gezeigten Schema.

○ Da nun das bc-Registerpaar die Position der Kugel enthält, muß auch der Rücksprung nicht mehr vorbereitet werden: Bauen Sie die erste Unterprogrammebene direkt in das Hauptprogramm ein. Lösungsvorschlag im nächsten Teil.

Wer die Aufgaben geschafft hat, oder wen sie geschafft haben, der darf sich jetzt etwas entspannen: Tippen Sie dazu das Basic-Programm von *Bild 5* ein. Mit diesem Programm wird das Kugel-Problem zum Kugel-Spiel, mit dem Stichwort „alle Neune“ (Aufruf des Maschinenprogramms in Zeile 140).

Klaus Herklotz

*(Wird fortgesetzt)*

| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | 2A | 0E | 40 | | | ld hl,(400E) |
| 4085 | CD | BB | 40 | | | call KUGEL |
| 4088 | 4D | | | | | ld c,l |
| 4089 | 44 | | | | | ld b,h |
| 408A | C9 | | | | | ret |
| | | | | | | |
| 408B | 06 | 0F | | | KUGEL | ld b,0F |
| 408D | 36 | 00 | | | HIN | ld (hl),00 |
| 408F | 23 | | | | | inc hl |
| 4090 | 36 | 34 | | | | ld (hl),34 |
| 4092 | 3A | 25 | 40 | | | ld a,(4025) |
| 4095 | FE | FF | | | | cp FF |
| 4097 | C0 | | | | | ret nz |
| 4098 | CD | B2 | 40 | | | call WARTE |
| 409B | 10 | F0 | | | | djnz HIN |
| 409D | 06 | 0F | | | | ld b,0F |
| 409F | 36 | 00 | | | RUECK | ld (hl),00 |
| 40A1 | 2B | | | | | dec hl |
| 40A2 | 36 | 34 | | | | ld (hl),34 |
| 40A4 | 3A | 25 | 40 | | | ld a,(4025) |
| 40A7 | FE | FF | | | | cp FF |
| 40A9 | C0 | | | | | ret nz |
| 40AA | CD | B2 | 40 | | | call WARTE |
| 40AD | 10 | F0 | | | | djnz RUECK |
| 40AF | C3 | 8B | 40 | | | jp KUGEL |
| | | | | | | |
| 40B2 | 11 | 00 | 03 | | WARTE | ld de,0300 |
| 40B5 | 1B | | | | LOOP | dec de |
| 40B6 | 7A | | | | | ld a,d |
| 40B7 | B3 | | | | | or e |
| 40B8 | 20 | FB | | | | jr nz,LOOP |
| 40BA | C9 | | | | | ret |

④ **Lösung des Kugel-Problems:** Oben das Hauptprogramm mit dem Aufruf der ersten Unterprogrammebene. Der Programmteil „WARTE“ ist die zweite Unterprogrammebene

```
100 REM BASIC-ERGAENZUNG
110 PRINT AT 21,4;">";TAB 21;"<
";AT 3,12;CHR$ 135;AT 4,11;CHR$
135;CHR$ 0;CHR$ 135;AT 5,10;CHR$
135;CHR$ 0;CHR$ 135;CHR$ 0;CHR$
135;AT 6,11;CHR$ 135;CHR$ 0;CHR
$ 135;AT 7,12;CHR$ 135;AT 21,5;
120 LET Z=0
130 FOR L=1 TO 2
140 LET Q=USR 16514
150 LET F=33
160 FOR N=1 TO 20
170 POKE Q,0
180 LET Q=Q-F
190 LET P=PEEK Q
200 IF P=135 THEN GOSUB 300
210 POKE Q,52
220 NEXT N
230 POKE Q,0
240 IF Z<9 THEN NEXT L
250 PRINT AT 21,9;Z;" TREFFER"
260 IF INKEY$="" THEN GOTO 260
270 GOTO 100
300 LET Z=Z+1
310 LET F=33+SGN INT (9*RND-4)
320 LET T=Q-66+F
330 IF PEEK T<>0 THEN LET Z=Z+1
340 POKE T,0
350 LET T=Q-132+2*F
360 IF PEEK T<>0 THEN LET Z=Z+1
370 POKE T,0
380 RETURN
```

⑤ **Spielprogramm:** Diese Ergänzung zum Maschinenprogramm zeigt, wie vorzüglich sich Basic- und Maschinenprogramme verknüpfen lassen

Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 15: Rettungskurs für 16-Bit-Daten

**Hochstapelei ist auch beim Retten von 16-Bit-Daten riskant. Sicherer sind Daten in reservierten Speicherzellen gelagert.**

Bevor es um das Retten von Daten geht, wollen wir noch die Übungen aus dem letzten Teil unter die Lupe nehmen. Gleich die erste Aufgabe dürfte einige Fragen aufgeworfen haben: Wenn nämlich ein relativer Sprung (Operationscode 18h) den absoluten Sprung ersetzt, so beträgt die Sprungweite DAh.

Ersetzt man indes die davon betroffene Bytefolge C38B40h durch 18DAh, dann wird die Bewegung der Kugel langsamer! Der Grund: Durch den relativen Sprung (im Listing z. B. als *jr ROUTIN* einzutragen) wird das Programm in der Adreßzeile 40AFh um ein Byte kürzer, und das Unterprogramm „WARTE" beginnt bereits mit Adresse 40B1h. Berücksichtigt man das nicht, dann wird die Warteschleife nach wie vor unter Adresse 40B2h aufgerufen – dort steht jetzt aber schon das zweite Byte (00) des Unterprogramms. Es ist reine Glückssache, daß durch diesen Einsprung das Programm nicht „abstürzt", sondern nur langsamer wird (durch Zufall hoher Wert im Register de).

Bei der zweiten Aufgabe mußte man die beiden *djnz*-Befehle durch *dec b* und *jr nz HIN* bzw. *jr nz RUECK* ersetzen. Die Bytefolge 10F0h ist also in 0520EFh umzuwandeln. Dabei ist die Änderung der Sprungweite schon einkalkuliert.

Aber auch hier sind die Folgen gravierend: Das Datenbyte (Sprungweite) des relativen Sprungs *jr ROUTIN* hat nicht mehr den Wert DAh, sondern D8h! Die Verzögerungsschleife „WARTE" ist nunmehr ab Adresse 40B3h zu finden.

Die dritte Aufgabe war nur eine Vorstufe zur vierten Aufgabe: Beide Aufgaben klärt der Lösungsvorschlag für die vierte Aufgabe (Listing auf Seite 74). Weil es keinen Befehl der Art *ld (bc), N* gibt, muß der Akkumulator als zwischenzeitlicher Datenträger herhalten: Die Befehle *xor a* (das entspricht *ld a, 00*) bzw. *ld a, 34* bringen das erwünschte Datenbyte (Kugel löschen bzw. schreiben) in den Akku, und *ld (bc), a* lädt es dann in die entsprechende Bildspeicherzeile. Ansonsten hält sich das Listing eng an das aus dem letzten Heft.

### Speichermethoden für 16-Bit-Daten

Beim Kugel-Programm ging alles relativ glatt über die Bühne, denn die Anzahl der verfügbaren Register reichte völlig aus. Sollen aber einmal größere Probleme bewältigt werden, so können leicht Zwangssituationen entstehen. Deshalb werden wir jetzt Schwierigkeiten, wie sie allgemein beim Speichern auftreten können, aufdecken und beseitigen.

Das wohl einfachste Verfahren, 16-Bit-Daten zu retten, bieten die beiden Befehle *push* und *pop*: *push rp* wirft eine Kopie des Registerpaares rp auf den Stapel, während *pop rp* den obersten 16-Bit-Zahlenwert vom Stapel abnimmt und ihn ins Registerpaar rp lädt. Diese Methode, Daten zu speichern, ist einfach, eignet sich aber nur für kurzfristiges Speichern.

Sollte der Speicherplatz einmal sehr knapp werden, das heißt, sollten so ziemlich alle Register belegt sein, dann können Austauschbefehle weiterhelfen: Der Befehl *ex de, hl* mit dem Operationscode EBh vertauscht den Inhalt der Registerpaare de und hl. Dagegen wirft der Befehl *ex (sp), hl* mit dem Operationscode E3h eine Kopie des hl-Registerpaares auf den Stapel; er hebt zuvor aber den obersten 16-Bit-Wert vom Stapel ab und lädt ihn ins hl-Registerpaar. Dieser nützliche Befehl vertauscht also die 16-Bit-Daten an der Oberfläche des Stapels mit dem Inhalt

| Mnemonic | Operationscode | symbolische Erläuterung |
|---|---|---|
| ld sp, NN | 31 | sp ← NN |
| ld sp, (NN) | ED7B | $sp_H$ ← (NN+1); $sp_L$ ← (NN) |
| ld (NN), sp | ED73 | (NN+1) ← $sp_H$; (NN) ← $sp_L$ |
| ld sp, hl | F9 | sp ← hl |
| inc sp | 33 | sp ← sp+1 |
| dec sp | 3B | sp ← sp−1 |
| add hl, sp | 39 | hl ← hl+sp |

① **Stapelzeiger als Register:** Diese Befehle ermöglichen ein Durchwühlen des Stapels (stack). Das sp-Register ist dabei einem gewöhnlichen Registerpaar im Datentransport und in der Arithmetik ebenbürtig

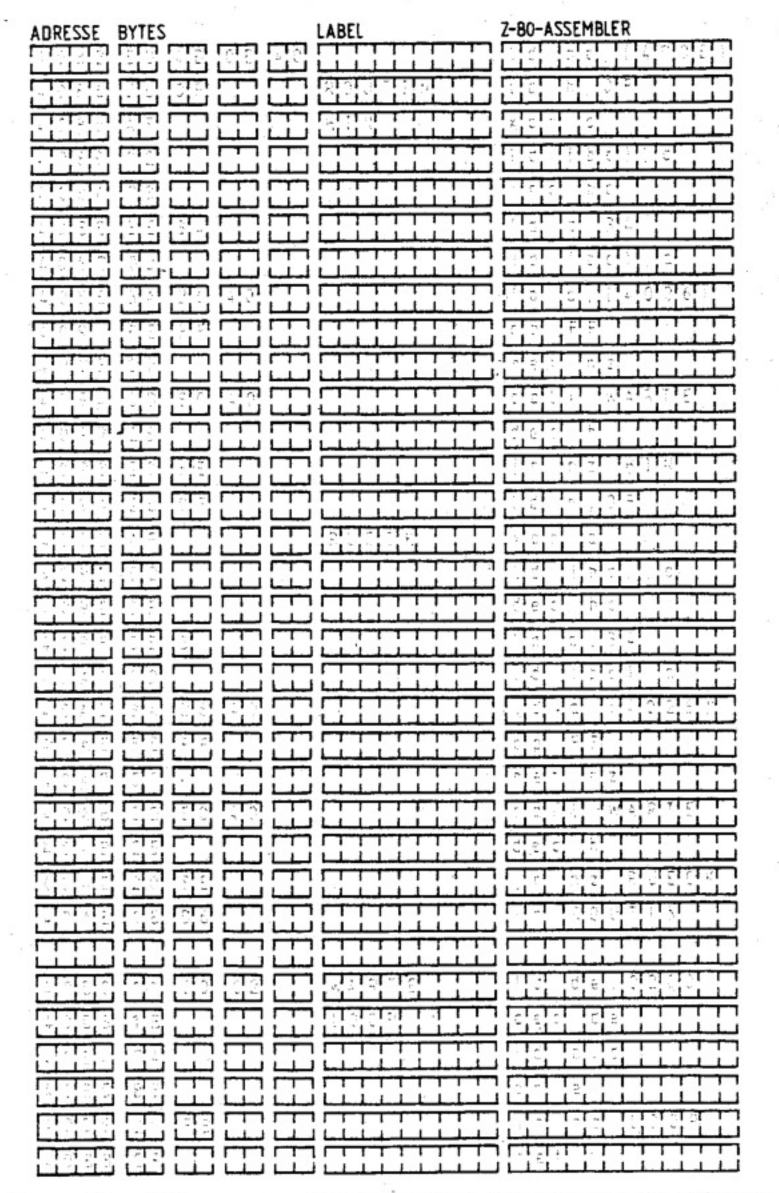
des hl-Registerpaares. Der Stand des Stapelzeigers bleibt dabei unverändert!

Zu ernsten Schwierigkeiten kommt es, wenn sich Daten, die man vom Stapel abnehmen will, nicht mehr an dessen Oberfläche befinden. Es gibt zwar Befehle, mit denen man im Stapel „herumwühlen" kann (*Bild* 1), vor deren Verwendung sei aber gewarnt! Es kann allzuleicht passieren, daß der Prozessor dann durch *ret* nicht mehr die Rücksprungadresse erhält, sondern die Daten eines abgespeicherten Registerpaares abhebt. Die Folgen davon kann sich jeder ausdenken.

Die wohl sicherste Möglichkeit, Daten abzuspeichern, bieten 16-Bit-Ladebefehle. Es müssen dazu lediglich zu Beginn des Programms eine Anzahl von Speicherzellen durch *nop's* freigehalten werden. Im Verlauf des Programms können 16-Bit-Daten dort durch *ld (NN), rp* gespeichert bzw. durch *ld rp, (NN)* geladen werden.

Klaus Herklotz

*(Wird fortgesetzt)*

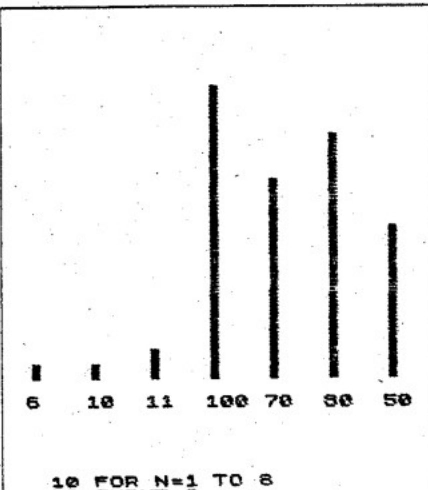



**Neues Kugel-Programm:** Sobald das bc-Registerpaar die Adresse der angepeilten Bildspeicherzelle erhält, wird das Programm um einige Bytes kürzer

## ZX-81-Software:

# Säulendiagramm in zwei Minuten

Säulendiagramme sind ein sehr übersichtliches Verfahren, um zu vergleichende Zahlenangaben grafisch auf dem Bildschirm darzustellen. Bei Wahlsendungen zeigen Säulendiagramme z. B. anschaulich die Gewinne und Verluste einzelner Parteien an einer Prozentskala und bei Haushaltsprogrammen können Säulendiagramme den Anteil einzelner Posten am Gesamtetat aufzeigen.




```
  10 FOR N=1 TO 8
  20 INPUT A
  30 IF A<0 OR A>100 THEN
GOTO 20
40 PRINT AT 21,4*(N-1);
INT A
  50 FOR M=1 TO 20
  60 IF A/5-M>0 THEN PRINT
AT M- 20,4*(N-1);"█"
  70 NEXT M
  80 NEXT N
 100 GOTO 100
```

**Säulenprogramm:** Das Grafiksymbol in Zeile 60 ist auf Taste 8 zu finden. Die Säulen werden im 5-%-Raster aufgebaut

Das hier vorgestellte Programm (*Bild*) ist in zwei Minuten eingetippt. Es zeigt nach der Eingabe eines Prozentwerts den Wert und die zugehörige Säule an. Maximal acht Säulen werden abgebildet, wobei die Höhenstufung in 5-%-Schritten erfolgt. Das Bild bleibt wegen Zeile 100 so lange ohne Meldung in der untersten Bildschirmzeile erhalten, bis mit BREAK abgebrochen wird.

Wolfgang Götz

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 16: Rechnen mit dem Carry-Flag

**Beinahe so wie ein ABC-Schütze notiert sich auch die Z-80-CPU mit dem Sprüchlein „Eins gemerkt" einen Übertrag, und zwar im Carry-Flag.**

Erinnern wir uns: Alle Flags befinden sich im f-Register (siehe Teil 11), und eines der Flags, das Zero- oder Null-Flag, haben wir schon unter die Lupe genommen. Es wird immer dann gesetzt, wenn das Ergebnis einer arithmetischen oder logischen Operation Null ist.

### So kommt das Carry-Flag ins Spiel

Das zweite, wichtige Flag heißt Carry-Flag (C-Flag); oft wird es auch als Übertrag-Flag bezeichnet. Das Carry-Flag wird immer nur dann gesetzt, wenn eine arithmetische Operation einen Übertrag vom höchstwertigen Bit des Operanden oder des Ergebnisses bewirkt.

Ein Übertrag vom höchstwertigen Bit des Ergebnisses kann bei einer Addition vorkommen. Dies ist z. B. dann der Fall, wenn durch *add a, FF* das Datenbyte FFh zum (hypothetischen) Akkumulatorinhalt DDh addiert wird (b markiert nachfolgend die binäre Schreibweise):

$$
\begin{aligned}
\text{DDh} + \text{FFh} &= 1101\ 1101\text{b} \\
&+ \underline{1111\ 1111\text{b}} \\
&1\ 1101\ 1100\text{b} = 1\text{DCh}
\end{aligned}
$$

Wie aus dem Rechengang hervorgeht, paßt das vorderste Bit des Ergebnisses nicht mehr in den Akku und wandert deshalb ins Carry-Flag. Nachdem diese Operation durchgeführt wurde, steht im Akku die Zahl DCh und das C-Flag ist gesetzt. Sollte sich bei der nächsten Operation kein Übertrag ergeben, so wird das C-Flag automatisch rückgesetzt!

Ein Übertrag vom höchstwertigen Bit des Operanden kann bei einer Subtraktion notwendig sein. Dies ist z. B. dann der Fall, wenn *sub a, FF* das Datenbyte FFh vom Akkumulatorinhalt DDh subtrahiert:

$$
\begin{aligned}
\text{DDh} - \text{FFh} &= 1\ 1101\ 1101\text{b} \\
&- \underline{1111\ 1111\text{b}} \\
&1101\ 1110\text{b} = \text{DEh}
\end{aligned}
$$

Der Rechengang zeigt diesmal, daß der Minuend (erste Zahl) kleiner als der Subtrahend wäre: Der Prozessor muß sich also einen Übertrag ausleihen, um die Operation überhaupt durchführen zu können; er setzt deshalb das Carry-Flag. Sollte auch hier einmal kein Übertrag vorliegen, dann wird das C-Flag rückgesetzt.

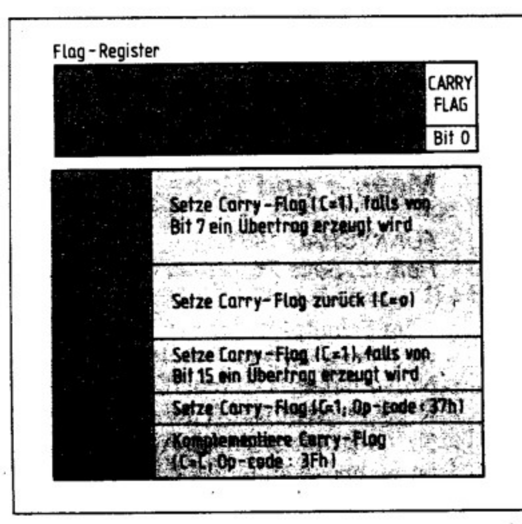
Alle arithmetischen Operationen setzen das Carry-Flag in Abhängigkeit von einem dabei auftretenden Übertrag. Alle logischen Operationen setzen das C-Flag dagegen prinzipiell zurück (*Bild 1*). Zusätzlich gibt es noch zwei Befehle, die zu den Einzelbitbefehlen zählen: *scf* setzt das C-Flag, während *ccf* das C-Flag invertiert.

### Arithmetik mit dem Carry-Flag

Genau wie das Zero-Flag, bliebe auch das Carry-Flag alleine ziemlich wirkungslos. In der Z-80-Maschinensprache gibt es daher eine Vielzahl von Befehlen, für die der Zustand des Carry-Flags von Bedeutung ist.

Da sind zuerst einmal die Sprungbefehle (*Bild 2*, oben). Mit der Sprungbedingung *c* erfolgt ein Sprung nur dann, wenn das C-Flag gesetzt ist. Umgekehrt erfolgt ein Sprung mit der Bedingung *nc* nur dann, wenn das C-Flag nicht gesetzt ist. Darüber hinaus gibt es spezielle Arithmetikbefehle, die auf das Übertrag-Flag zurückgreifen.

Stellen wir uns vor, es sollen zwei 32-Bit-Zahlen addiert werden. Unsere bisherigen 16-Bit-Additionsbefehle reichen dann gerade für die hintere Hälfte. Wir wissen aber, daß ein Arithmetikbefehl einen Übertrag ins Carry-Flag bringt. Nützlich wäre deshalb ein weiterer Additionsbefehl, der das Carry-Flag berücksichtigt: Sobald bei der letzten Operation ein Übertrag entstan-




① **Carry-Flag:** Dieses Flag kann gesetzt, zurückgesetzt oder komplementiert werden und es ist vom Ergebnis arithmetischer Operationen abhängig (s ≙ 8-Bit-Zahl)

| | | |
|---|---|---|
| jp c NN | DA .. .. | jp NN falls CY = 1 |
| jp nc NN | D2 .. .. | jp NN falls CY = 0 |
| jr c DIS | 38 .. | jr DIS falls CY = 1 |
| jr nc DIS | 30 .. | jr DIS falls CY = 0 |
| call c NN | DC .. .. | call NN falls CY = 1 |
| call nc NN | D4 .. .. | call NN falls CY = 0 |
| ret c | D8 | ret falls CY = 1 |
| ret nc | D0 | ret falls CY = 0 |

| | | | |
|---|---|---|---|
| adc a, a | 8F | sbc a, a | 9F |
| adc a, b | 88 | sbc a, b | 98 |
| adc a, c | 89 | sbc a, c | 99 |
| adc a, d | 8A | sbc a, d | 9A |
| adc a, e | 8B | sbc a, e | 9B |
| adc a, h | 8C | sbc a, h | 9C |
| adc a, l | 8D | sbc a, l | 9D |
| adc a, (hl) | 8E | sbc a, (hl) | 9E |
| adc a, N | CE .. | sbc a, N | DE |
| adc hl, bc | ED 4A | sbc hl, bc | ED 42 |
| adc hl, de | ED 5A | sbc hl, de | ED 52 |
| adc hl, hl | ED 6A | sbc hl, hl | ED 62 |

② **Wirkung des Carry-Flags:** Spezielle Sprung- und Arithmetikbefehle berufen sich auf den Zustand des C-Flags. Sollte Verwechslungsgefahr mit dem c-Register bestehen, dann wird das Flag mit CY abgekürzt

den ist, soll automatisch 1 dazuaddiert werden.

Im Z-80-Befehlssatz sind die beiden Kürzel *adc* (Add with Carry) und *sbc* (Subtract with Carry) zu finden. Der Befehl *adc a, s* entspricht dem Befehl *add a, s*, nur daß er zusätzlich das Carry-Flag addiert (s ist z. B. irgendein Registerinhalt). Umgekehrt wird mit *sbc a, s* auch das Carry-Flag subtrahiert. Beide Befehle gibt es sowohl für 8-Bit-Zahlen, als auch für 16-Bit-Zahlen (Bild 2, unten).

Mit diesen neuen Arithmetikbefehlen lassen sich praktisch beliebig große Zahlen (in 8- oder 16-Bit-Portionen aufgeteilt) addieren und subtrahieren. Schwierigkeiten treten dann aber beim Ausgeben des Ergebnisses auf: Mit dem bc-Registerpaar läßt sich höchstens der Dezimalwert einer 16-Bit-Zahl in ein Basic-Programm mitschleppen. Gezwungenermaßen können wir deshalb nur begrenzt Rechenoperationen durchführen.

*Bild 3* zeigt oben ein Maschinenprogramm, das auf herkömmliche Weise zwei 16-Bit-Zahlen addiert und das Ergebnis über das bc-Registerpaar dem Basic meldet. Der 16-Bit-Additionsbefehl soll nun durch zwei 8-Bit-Befehle ersetzt werden. Wie das zweite Listing in Bild 3 unten zeigt, laufen beide 8-Bit-Additionen über den Akkumulator ab: Bei der zweiten Addition (Adresse 408Ch) kommt dann das Carry-Flag zum Einsatz. Anderenfalls bliebe ein von der ersten Addition herrührender Übertrag unerkannt. Klaus Herklotz

*(Wird fortgesetzt)*

| ADRESSE | BYTES | | | | Z-80-ASSEMBLER |
|---|---|---|---|---|---|
| 4082 | 21 | 22 | 11 | | ld hl, SUM1 |
| 4085 | 01 | 44 | 33 | | ld bc, SUM2 |
| 4088 | 09 | | | | add hl, bc |
| 4089 | 4D | | | | ld c, l |
| 408A | 44 | | | | ld b, h |
| 408B | C9 | | | | ret |
| | | | | | |
| 4082 | 21 | 22 | 11 | | ld hl, SUM1 |
| 4085 | 01 | 44 | 33 | | ld bc, SUM2 |
| 4088 | 79 | | | | ld a, c |
| 4089 | 85 | | | | add a, l |
| 408A | 4F | | | | ld c, a |
| 408B | 78 | | | | ld a, b |
| 408C | 8C | | | | adc a, h |
| 408D | 47 | | | | ld b, a |
| 408E | C9 | | | | ret |

③ **Addieren zweier 16-Bit-Zahlen:** Oben addiert ein 16-Bit-Befehl die beiden Summanden 1122h und 3344h, unten erfüllen zwei 8-Bit-Befehle dieselbe Aufgabe. PRINT USR 16514 bringt das Ergebnis als Dezimalzahl (17510) auf den Bildschirm

Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 17: Schieben und Rotieren

**Schieberegister gibt's fix und fertig als Hardware. Mit speziellen Maschinenbefehlen lassen sich Bits aber auch in den Registern der Z-80A-CPU hin- und herschieben. Per Befehl kommen die Bits sogar ins Rotieren.**

Ein Problem, mit dem sich früher oder später jeder Programmierer auseinandersetzen muß, ist das Ausgeben von Zahlenwerten auf dem Bildschirm. Denn wie will man sonst das Ergebnis einer Rechenoperation erfahren?

Es müßte uns zumindest gelingen, eine Hex-Zahl, die sich gerade im Akkumulator befindet, auf den Bildschirm zu bringen. So einfach geht dies aber nicht! Jede 8-Bit-Zahl besteht nämlich im hexadezimalen Zahlensystem aus zwei Ziffern. Und von diesen beiden Ziffern müssen wir zuerst den Code (siehe Sinclair-Handbuch, Anhang A) errechnen, um ihn in zwei aufeinanderfolgende Bildspeicherzellen laden zu können.

### Aufspalten einer Hex-Zahl

Bei der linken Ziffer (*Bild 1*) müssen zunächst alle Bits um vier Stellen nach rechts geschoben werden. Nur so bringt man diese Ziffer allein in ein Register (hier: aus 9Ch wird 09h). Damit der Code der Ziffer vorliegt, braucht zu dieser bloß noch 1Ch addiert werden (hier: 09h + 1Ch = 25h). Die Addition von 1Ch ≙ 28 ist notwendig, weil im Zeichenvorrat des ZX 81 die Ziffern ab der Code-Nummer 28 abgelegt sind. Das Ergebnis (hexadezimal: 25; dezimal: 37 ≙ Code für „9") kann nun in die erste Bildspeicherzelle geladen werden, und die linke Ziffer (9) steht am Bildschirm.

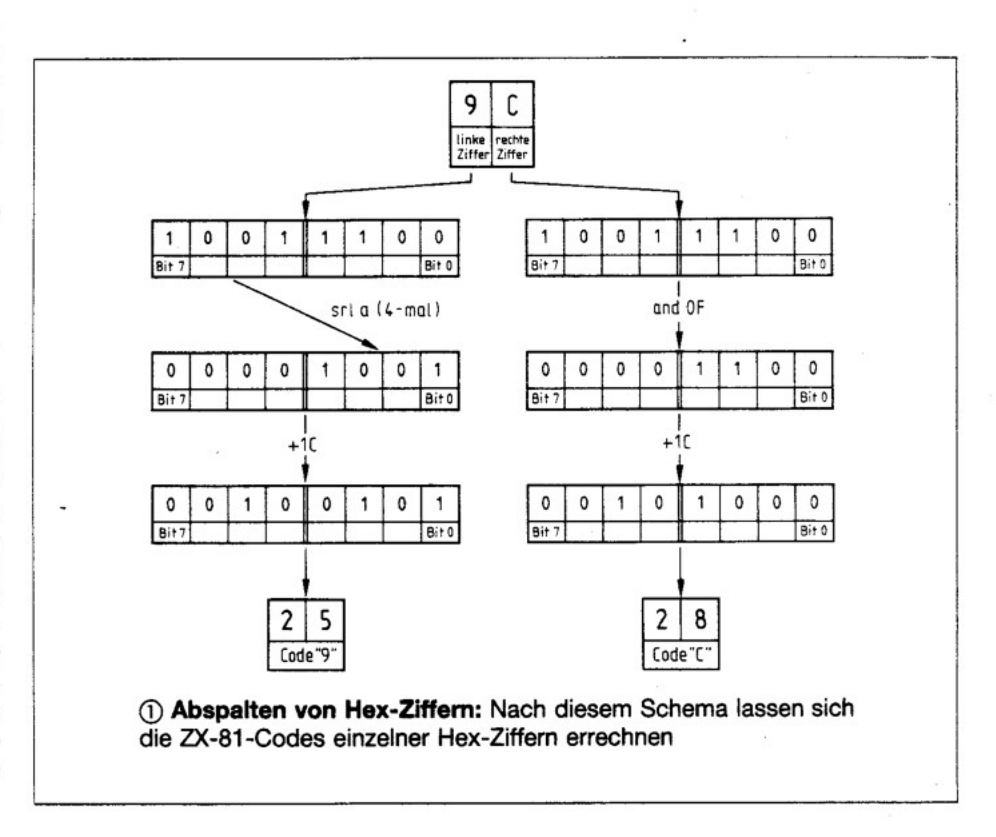
Bei der rechten Ziffer verfährt man ähnlich: Zuerst ist deren linke Hälfte (nachfolgend auch linkes Nibble genannt) auszublenden. Dies bewirkt grundsätzlich der Befehl *and 0F* (Merke: 9Ch *and* 0F = 0Ch, siehe Teil 10). Danach kann wieder 1Ch addiert werden. Das Ergebnis ist der Code der zweiten Ziffer (C), der in die zweite Bildspeicherzelle zu laden ist.

Ein Problem ergibt sich bei der ganzen Angelegenheit: Wie schiebt man am besten das linke Nibble nach rechts? Z-80-Rotations- und -Schiebefehle sind dafür reichhaltig vorhanden (*Bild 2*).

Im Prinzip werden bei den Schiebebefehlen alle Bits des Operanden um eine Binärstelle verschoben! Der *sla*-Befehl lädt dabei Bit 0 mit einer logischen Null; Bit 7 wandert ins Carry-Flag. Genau umgekehrt verhält es sich beim *srl*-Befehl. Beim *sra*-Befehl behält lediglich Bit 7 seinen Wert. Die Richtung der Verschiebung legt der zweite Buchstabe im Mnemonik-Kürzel fest (r für rechts und *l* für links).

Bei den Rotationsbefehlen wandern die Bits kreisförmig in einem Register. Daher legt der zweite Buchstabe im Mnemonik-Kürzel die Drehrichtung fest. Der *rlc*-Befehl (c für circular) bringt dabei Bit 7 des Operanden ins Carry-Flag, wogegen der *rl*-Befehl das Carry-Bit vom Flag-Register gleich als neuntes Bit verwendet. Die restlichen beiden Befehle *rrc* und *rr* führen das gleiche in der anderen Drehrichtung durch.

Weiterhin gibt es noch zwei komplexere Rotierbefehle: *rld* und *rrd* vertauschen gleich eine ganze Ziffer zwischen dem Akkumulator und einer durchs hl-Registerpaar festgelegten Speicherstelle. Das linke Nibble des Akkumulators bleibt davon ausgeschlossen.




① **Abspalten von Hex-Ziffern:** Nach diesem Schema lassen sich die ZX-81-Codes einzelner Hex-Ziffern errechnen

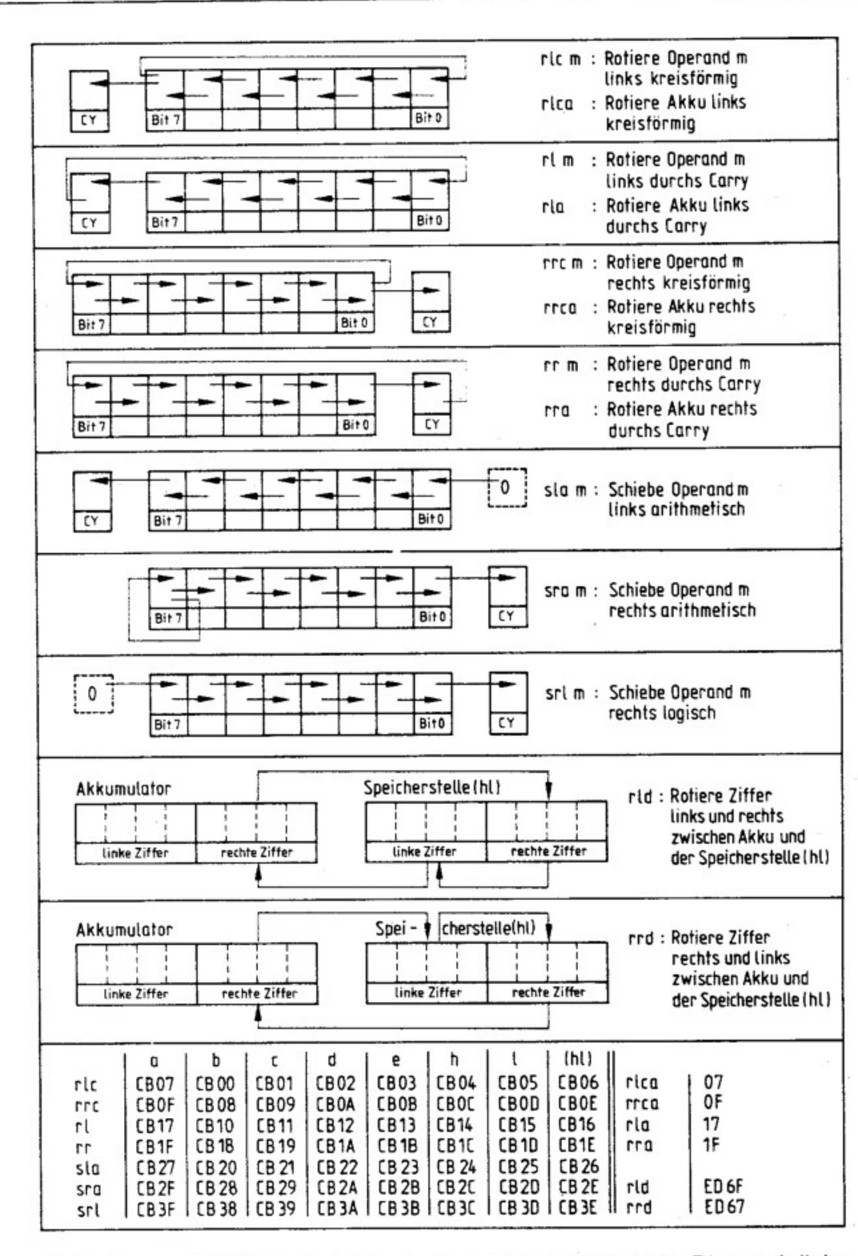
## Na endlich: hexadezimale Ausgabe

Am nützlichsten für unsere Zwecke ist der Befehl *srl a*: Schiebe die Bits des Akkumulators nach rechts. Dabei wird das höchstwertige Bit auf logisch Null gesetzt. Mit Hilfe dieses Befehls entstand auch das Listing von *Bild 3*.

Mit dem Programm wird im ersten Schritt die Adresse der ersten Bildspeicherzelle ins de-Registerpaar gebracht (4082h bis 4086h) und der Akku mit der auszugebenden Zahl geladen (4087h). Um den Inhalt des Akkus zu retten, legt *push af* eine Kopie des af-Registerpaares auf den Stapel. Anschließend werden die Bits des Akkumulators viermal nach rechts geschoben (408Ah bis 4091h) und zum Ergebnis wird 1Ch addiert (4092h). Bei Adresse 4094h wird die linke Ziffer – in Form dieses Ergebnisses – auf den Bildschirm gebracht.

Im zweiten und letzten Schritt erhält das de-Registerpaar die Adresse der zweiten Bildspeicherzelle (4095h). *pop af* stellt die früheren Zustände im Akkumulator wieder her (4096h) und eine logische UND-Verknüpfung blendet die linke Ziffer aus (4097h). Abschließend wird auch hier 1Ch addiert und die Ziffer auf den Bildschirm gebracht (409Bh).

Mit Hilfe dieses Programmes lassen sich alle Register und Speicherzelleninhalte – sofern sie in den Akku kopiert wurden – innerhalb eines Maschinenprogramms ausgeben. In unserem Fall (Bild 3) bringt RAND USR 16514 den Akkuinhalt 9Ch auf den Bildschirm.




| | a | b | c | d | e | h | l | (hl) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| rlc | CB07 | CB00 | CB01 | CB02 | CB03 | CB04 | CB05 | CB06 | rlca | 07 |
| rrc | CB0F | CB08 | CB09 | CB0A | CB0B | CB0C | CB0D | CB0E | rrca | 0F |
| rl | CB17 | CB10 | CB11 | CB12 | CB13 | CB14 | CB15 | CB16 | rla | 17 |
| rr | CB1F | CB18 | CB19 | CB1A | CB1B | CB1C | CB1D | CB1E | rra | 1F |
| sla | CB27 | CB20 | CB21 | CB22 | CB23 | CB24 | CB25 | CB26 | | |
| sra | CB2F | CB28 | CB29 | CB2A | CB2B | CB2C | CB2D | CB2E | rld | ED6F |
| srl | CB3F | CB38 | CB39 | CB3A | CB3B | CB3C | CB3D | CB3E | rrd | ED67 |

② **Rotier- und Schiebebefehle:** Im Operanden werden die Bits nach links, nach rechts oder im Kreis verschoben

Noch ein paar Worte zu den Rotations- und Schiebebefehlen: Im Grunde müßten die Befehle *rlca* (Opcode 07h) und *rlc a* (Opcode CB07) identisch sein. Dem ist aber nicht so. Beide Befehle beeinflussen die Flags unterschiedlich. So bleibt z. B. das Zero-Flag vom Befehl *rlca* unbeeinflußt, während *rlc a* das Zero-Flag in Abhängigkeit vom Ergebnis setzt bzw. zurücksetzt. Ein weiterer Unterschied ist die Verarbeitungszeit: Der Befehl *rlca* wird doppelt so schnell ausgeführt.

## ZX 81 als Hex-Rechner

Carry-Flag und Schiebebefehle ermöglichen erstmals die Ausgabe von Hex-Daten. Wir wollen das gleich in der Praxis erproben und eine Routine erstellen, die zwei 24-Bit-Hexadezimalzahlen addiert und das Ergebnis auf den Bildschirm schreibt.

Zuallererst stellt sich dann die Frage, wie man beide Summanden eingeben soll. Zweckmäßig ist es, am Anfang des Programmes einfach zehn Speicherzellen für Daten zu reservieren: die ersten sechs für die beiden 24-Bit-Summanden, die letzten vier für das 25-Bit-Ergebnis (Übertrag). Somit können die beiden Summanden direkt am Anfang des Strings A$ mit dem Maschinenprogramm eingegeben werden: Die Anweisung

LET A$="AAAAAABBBBBB00000000..."

legt also fest, daß die beiden Hex-Zahlen AAAAAAh und BBBBBBh addiert werden, wobei das Ergebnis in die vier durch Nullen reservierte Speicherzellen

abgelegt wird. Nach Abschluß des Rechenvorgangs bringt das bekannte Unterprogramm WRITE das Ergebnis auf den Bildschirm.

Die Bytes des Maschinenlistings aus *Bild 4* sind wie immer für eine REM-Zeile gedacht: LET Q=USR 16524 bringt das Programm nach dem POKE-Vorgang zum Laufen.

Am Anfang des Programms reservieren wie vereinbart *nop*-Befehle zehn Speicherzellen (4082h bis 408Bh). Bei Adresse 408Ch erhält das hl-Registerpaar die Adresse des letzten Bytes vom zweiten Summanden zugewiesen. Der Akkumulator wird mit dem letzten Byte des ersten Summanden geladen (408Fh) und dazu das letzte Byte des zweiten Summanden addiert (4092h). Anschließend erhält die letzte der vorher reservierten Speicherzellen das Ergebnis dieser Operation, das gleichzeitig das letzte Byte des Gesamtergebnisses ist (4093h).

In den Programmteilen von Adresse 4096h bis 409Dh und von 409Eh bis 40A5h wird ein weiteres Byte des Gesamtergebnisses berechnet. Dies geschieht wie beim eben beschriebenen Vorgang. Einziger Unterschied ist der *adc*-Befehl, der einen Übertrag vom letzten Byte berücksichtigt (409Ah und 40A2h).

Der Programmteil von Adresse 40A6h bis 40ABh dient einzig und allein dazu, den Übertrag vom 24sten Bit zu ermitteln und ihn in Form einer 0 oder 1 abzuspeichern. Im weiteren Verlauf des Programms werden die vier Bytes des Ergebnisses über eine Schleife ausgegeben (40ACh bis 40BCh). Das Unterprogramm WRITE bringt dabei die Ziffern auf den Bildschirm. Klaus Herklotz

(*Wird fortgesetzt*)

| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | 00 | 00 | 00 | | | nop |
| 4085 | 00 | 00 | 00 | | | nop |
| 4088 | 00 | 00 | 00 | 00 | | nop |
| 408C | 21 | 87 | 40 | | START | ld hl, 4087 |
| 408F | 3A | 84 | 40 | | | ld a, (4084) |
| 4092 | 86 | | | | | add a, (hl) |
| 4093 | 32 | 8B | 40 | | | ld (408B), a |
| 4096 | 2B | | | | | dec hl |
| 4097 | 3A | 83 | 40 | | | ld a, (4083) |
| 409A | 8E | | | | | adc a, (hl) |
| 409B | 32 | 8A | 40 | | | ld (408A), a |
| 409E | 2B | | | | | dec hl |
| 409F | 3A | 82 | 40 | | | ld a, (4082) |
| 40A2 | 8E | | | | | adc a, (hl) |
| 40A3 | 32 | 89 | 40 | | | ld (4089), a |
| 40A6 | 3E | 00 | | | | ld a, 00 |
| 40A8 | 8F | | | | | adc a, a |
| 40A9 | 32 | 88 | 40 | | | ld (4088), a |
| 40AC | ED | 5B | 0C | 40 | | ld de, (400C) |
| 40B0 | 13 | | | | | inc de |
| 40B1 | 21 | 88 | 40 | | | ld hl, 4088 |
| 40B4 | 06 | 04 | | | | ld b, 04 |
| 40B6 | 7E | | | | LOOP | ld a, (hl) |
| 40B7 | CD | BE | 40 | | | call WRITE |
| 40BA | 23 | | | | | inc hl |
| 40BB | 10 | F9 | | | | djnz LOOP |
| 40BD | C9 | | | | | ret |
| 40BE | F5 | | | | WRITE | push af |
| 40BF | CB | 3F | | | | srl a |
| 40C1 | CB | 3F | | | | srl a |
| 40C3 | CB | 3F | | | | srl a |
| 40C5 | CB | 3E | | | | srl a |
| 40C7 | C6 | 1C | | | | add a, 1C |
| 40C9 | 12 | | | | | ld (de), a |
| 40CA | 13 | | | | | inc de |
| 40CB | F1 | | | | | pop af |
| 40CC | E6 | 0F | | | | and 0F |
| 40CE | C6 | 1C | | | | add a, 1C |
| 40D0 | 12 | | | | | ld (de), a |
| 40D1 | 13 | | | | | inc de |
| 40D2 | C9 | | | | | ret |

④ **Rechnen mit Hex-Zahlen:** Der ZX-81 addiert eine in den ersten drei Speicherzellen abgelegte Hex-Zahl zu einer in den zweiten drei Speicherzellen abgelegten Hex-Zahl

| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | ED | 5B | 0C | 40 | | ld de, (400C) |
| 4086 | 13 | | | | | inc de |
| 4087 | 3E | 9C | | | | ld a, 9C |
| 4089 | F5 | | | | | push af |
| 408A | CB | 3F | | | | srl a |
| 408C | CB | 3F | | | | srl a |
| 408E | CB | 3F | | | | srl a |
| 4090 | CB | 3F | | | | srl a |
| 4092 | C6 | 1C | | | | add a, 1C |
| 4094 | 12 | | | | | ld (de), a |
| 4095 | 13 | | | | | inc de |
| 4096 | F1 | | | | | pop af |
| 4097 | E6 | 0F | | | | and 0F |
| 4099 | C6 | 1C | | | | add a, 1C |
| 409B | 12 | | | | | ld (de), a |
| 409C | C9 | | | | | ret |

③ **Ausgabe von Hex-Zahlen:** Das Maschinenprogramm bringt den Akkumulatorinhalt an den Tag

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 18: Der Speicher wird transparent

**Das folgende Anwendungsbeispiel für Maschinensprache deckt schlagartig den Inhalt von 176 Speicherzellen auf.**

Geben Sie zuerst die Bytes des Listungs (*Bild*) mit einem Eingabeprogramm ( siehe Teil 3 oder Teil 11) ein. Dann ist das Maschinenprogramm mit dem Kommando

LET Q=USR 16514

zu starten: Der Bildschirm zeigt daraufhin den Inhalt von 176 Zellen des Speicherbereichs: Vorneweg steht in jeder Zeile die Adresse der ersten Speicherzelle, gefolgt von acht Bytes.

Die Adresse der ersten Speicherzelle wird mit Hilfe der Systemvariablen SEED festgelegt, die unter den Adressen 16434 (4032h) und 16435 zu finden ist. Der Basic-Befehl RAND weist dieser Variablen direkt den auf RAND folgenden Zahlenwert zu (siehe Sinclair-Handbuch, Kapitel 28). 16-Bit-Daten können daher ohne viel Rechnerei bzw. „Pokerei" eingegeben werden. So bewirkt z. B. die Eingabe von

RAND 16514

vor dem Aufruf des Programmes, daß der ZX-81 den Inhalt des Speicherbereichs ab Adresse 16514 (4082h) ausgibt.

Das Registerpaar de bezieht zunächst, aus der Systemvariablen D-FILE die Adresse der ersten Bildspeicherzelle (4082h bis 4086h) und das hl-Registerpaar bekommt die Adresse des ersten auszudruckenden Bytes mit Hilfe von SEED zugewiesen (4087h).

Anschließend wird die erste Schleife festgelegt: Zählerregister b erhält den Startwert 16h, was der Anzahl der 22 Zeilen am Bildschirm entspricht. In jeder Zeile erscheint als erstes die Adresse. Nachdem der Akku mit dem jeweiligen Byte (erstes bzw. zweites) der Adresse geladen wurde (408Ch und 4090h), erfolgt ein Aufruf der Schreibroutine (408Dh und 4091h). Danach wird, damit etwas Raum zwischen der Adresse und den Bytes bleibt, das Register de zweimal inkrementiert.

Jetzt wird in die erste Schleife eine weitere Schleife verschachtelt; *push bc* wirft den alten Schleifenzählerstand auf den Stapel (4096h) und *ld b, 08* legt den neuen Startwert (Anzahl der Bytes pro Zeile) fest. In dieser Schleife erhält der Akku den Inhalt der auszugebenden Speicherzelle (4099h). Die Schreibroutine bringt auch diesen Wert auf den Bildschirm (409Ah). Dann werden die nächste Bildspeicherzelle sowie das nächste auszugebende Byte festgelegt (409Dh und 409Eh). Diese innere Schleife endet mit einem Sprung zum Ziel BYTE.

Sobald die innere Schleife abgearbeitet wurde, erhöht *inc de* den Inhalt des Bildspeicherzellen-Registers de nochmals um insgesamt Drei, damit die nächste Adresse am Anfang der nächsten Zeile erscheint. Der Befehl *pop bc* holt dann den alten Schleifenzählerstand für die äußere Schleife zurück (40A4h) und die nächste Zeile am Bildschirm kann in Angriff genommen werden. So geht das weiter, bis letztendlich alle Daten ausgegeben sind.

Klaus Herklotz

(*Wird fortgesetzt*)

| Adresse | Bytes | Label | Z80-Assembler |
|---|---|---|---|
| 4082 | ED 5B 0C 40 | | ld de, (400C) |
| 4086 | 13 | | inc de |
| 4087 | 2A 32 40 | | ld hl, (4032) |
| 408A | 06 16 | | ld b, 16 |
| 408C | 7C | ZEILE | ld a, h |
| 408D | CD A8 40 | | call WRITE |
| 4090 | 7D | | ld a, l |
| 4091 | CD A8 40 | | call WRITE |
| 4094 | 13 | | inc de |
| 4095 | 13 | | inc de |
| 4096 | C5 | | push bc |
| 4097 | 06 08 | | ld b, 08 |
| 4099 | 7E | BYTE | ld a, (hl) |
| 409A | CD A8 40 | | call WRITE |
| 409D | 13 | | inc de |
| 409E | 23 | | inc hl |
| 409F | 10 F8 | | djnz BYTE |
| 40A1 | 13 | | inc de |
| 40A2 | 13 | | inc de |

| Adresse | Bytes | Label | Z80-Assembler |
|---|---|---|---|
| 40A3 | 13 | | inc de |
| 40A4 | C1 | | pop bc |
| 40A5 | 10 E5 | | djnz ZEILE |
| 40A7 | C9 | | ret |
| 40A8 | F5 | WRITE | push af |
| 40A9 | CB 3F | | srl a |
| 40AB | CB 3F | | srl a |
| 40AD | CB 3F | | srl a |
| 40AF | CB 3F | | srl a |
| 40B1 | C6 1C | | add a, 1C |
| 40B3 | 12 | | ld (de), a |
| 40B4 | 13 | | inc de |
| 40B5 | F1 | | pop af |
| 40B6 | E6 0F | | and 0F |
| 40B8 | C6 1C | | add a, 1C |
| 40BA | 12 | | ld (de), a |
| 40BB | 13 | | inc de |
| 40BC | C9 | | ret |

**Blick in den Speicher:** Dieses Maschinenprogramm schreibt im Nu den Inhalt von 176 Speicherzellen auf den Bildschirm. Ein Trick: Die Startadresse läßt sich mit RAND ohne „Pokerei" und Umrechnerei dezimal eingeben

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 19: Flaggenparade

**Jetzt nur nicht schlappmachen, auch wenn die vier noch verbliebenen Z-80-Flags nicht so bedeutend sind, wie das Zero- und Carry-Flag.**

Mit dem Zero- und Carry-Flag (siehe Teil 11 und Teil 16) kennen wir bereits die beiden wichtigsten Z-80-Flags. Wie die Übersicht im *Bild* zeigt, gibt es im f-Register darüber hinaus noch vier weitere Flags, die allerdings im Programmieralltag keine überragende Rolle spielen. Sie werden deshalb anschließend nur kurz beschrieben.

### Die restlichen Flags

Das Half-Carry-Flag (Halbübertrag-Flag; H-Flag) ist dem Carry-Flag sehr ähnlich. Bekanntlich zeigt das Carry-Flag einen Übertrag von Bit 7 an. Das Half-Carry-Flag zeigt dagegen einen Übertrag von Bit 3 nach Bit 4 (bei Additionen) bzw. ein Ausleihen von Bit 4 nach Bit 3 (bei Subtraktionen) an:
H = 1 falls ein Übertrag erzeugt wurde.
H = 0 falls kein Übertrag vorliegt.

Im Z-80-Befehlsvorrat gibt es keine Sprungbedingung, die vom Zustand dieses Flags abhängt. Das H-Flag wird vielmehr vom Prozessor selbst verwendet.

Das Add/Subtract-Flag (N-Flag) zeigt an, ob die letzte Operation eine Subtraktion war:
N = 1 falls die vorangegangene Operation (vorausgesetzt sie hat das N-Flag beeinflußt) eine Subtraktion war.
N = 0 falls die Operation keine Subtraktion war.
Für das N-Flag gibt es ebenfalls keine Sprungbedingungen.

Das „Parity or Overflow Flag" (Paritäts- oder Überlauf-Flag; P/V-Flag) erfüllt gleich zwei Aufgaben. Zum einen arbeitet es bei logischen, sowie bei Rotier- und Schiebeoperationen als Paritäts-Flag. Es zeigt also an, ob in der binären Darstellung des betroffenen Datenwortes eine gerade oder ungerade Anzahl von Einsen vorliegt:
P/V = 1 falls die Anzahl der Einsen gerade ist (gerade Parität).
P/V = 0 falls die Anzahl der Einsen ungerade ist (ungerade Parität).

Das P/V-Flag kann somit zur Kontrolle über die Unversehrtheit von Daten eingesetzt werden. Sollte sich z. B. wegen eines Übertragungsfehlers ein Bit eines Datenwortes verändert haben, so kann man dies mit Hilfe des P/V-Flags erkennen: Es muß lediglich die Parität (Gleichstellung) der Einsen überprüft werden.

Die jeweiligen Sprungbefehle zum P/V-Flag lauten:

| | | |
|---|---|---|
| jp pe, NN | EA | jp NN falls P/V = 1 (gerade Parität; parity even) |
| jp po, NN | E2 | jp NN falls P/V = 0 (ungerade Parität; parity odd) |
| call pe, NN | EC | call NN falls P/V = 1 |
| call po, NN | E4 | call NN falls P/V = 0 |
| ret pe | E8 | ret falls P/V = 1 |
| ret po | E0 | ret falls P/V = 0 |

Als weitere Funktion zeigt das P/V-Flag bei arithmetischen Operationen einen Überlauf an. Es ist nämlich in der Zweierkomplement-Darstellung (siehe Teil 12) einer Zahl üblich, daß das höchstwertige Bit dieser Zahl das Vorzeichen bestimmt: Liegt das höchstwertige Bit z. B. auf logisch 1, dann wird die Zahl als negative Zahl angesehen. Wird jetzt auf diese Art gerechnet, kann es leicht sein, daß ein Übertrag ins höchstwertige Bit wandert und das Vorzeichen verfälscht. Für arithmetische Operationen wurde deshalb eine Warneinrichtung vorgesehen:
P/V = 1 falls ein solcher Überlauf erzeugt wurde.
P/V = 0 falls kein solcher Überlauf erzeugt wurde.

Wie schon beim P/V-Flag erwähnt, markiert das höchstwertige Bit beim Zweierkomplement das Vorzeichen. Durch einige arithmetische und logische Operationen und durch Rotier- und Schiebebefehle wird dieses Bit ins Sign-Flag (Vorzeichen-Flag; S-Flag) kopiert:
S = 1 falls das Ergebnis der Operation negativ ist.
S = 0 falls das Ergebnis positiv ist.

Mit diesem Flag kommt es dann zu folgender Sprungpalette:

| | | |
|---|---|---|
| jp m, NN | FA | jp NN falls S = 1 (m weil minus) |
| jp p, NN | F2 | jp NN falls S = 0 (p weil positiv) |
| call m, NN | FC | call NN falls S = 1 |
| call p, NN | F4 | call NN falls S = 0 |
| ret m | F8 | ret falls S = 1 |
| ret p | F0 | ret falls S = 0 |

Zugegeben, bis hierher ist dieser Teil reichlich trocken und schwer verdaulich gewesen. Zur Erholung wollen wir deshalb wieder einmal Klartext im Detail machen: Diesmal geht es um die Einteilung der Z-80-Adressierungsarten, wie man sie häufig in der Literatur findet.

### Adressierung beim Wort genommen

Alle Z-80-Befehle lassen sich in Befehlsgruppen einordnen. Bezeichnend dafür ist das erste Mnemonic in der Assemblerschreibweise (Beispiele: *ld*, *jp*, *add*, usw.). Es kommt dann z. B. zu den Gruppen der Ladebefehle, der Sprungbefehle oder der Arithmetikbefehle.

Flag-Register

| SIGN FLAG | ZERO FLAG | unbelegt | HALF CARRY FLAG | unbelegt | P/V-FLAG | ADD/SUB. FLAG | CARRY FLAG |
|---|---|---|---|---|---|---|---|
| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |

| Befehl | S | Z | H | P/V | N | C | Erklärung |
|---|---|---|---|---|---|---|---|
| add a,s; adc a,s | ! | ! | ! | ! | 0 | ! | 8-bit Addition (mit CY) |
| sub a,s; sbc a,s; cp s; neg | ! | ! | ! | ! | 1 | ! | 8-bit Subtraktion (mit CY) Vergleich; Zweierkomplement |
| and s | ! | ! | 1 | ! | 0 | 0 | UND-Verknüpfung |
| or s; xor s | ! | ! | 0 | ! | 0 | 0 | (Exklusiv-) ODER-Verknüpfung |
| inc s | ! | ! | ! | ! | 0 | – | 8-bit Inkrement |
| dec s | ! | ! | ! | ! | 1 | – | 8-bit Decrement |
| add hl, ss | – | – | ? | – | 0 | ! | 16-bit Addition |
| adc hl, ss | ! | ! | ? | ! | 0 | ! | 16-bit Addition mit CY |
| sbc hl, ss | ! | ! | ? | ! | 1 | ! | 16-bit Subtraktion mit CY |
| rla; rlca; rra; rrca | – | – | 0 | – | 0 | ! | Rotation des Akkus |
| rl m; rlc m; rr m; rrc m sla m; sra m; srl m | ! | ! | 0 | ! | 0 | ! | Rotation Verschiebung |
| rld; rrd | ! | ! | 0 | ! | 0 | – | Ziffernrotation |
| cpl | – | – | 1 | – | 1 | – | Einerkomplement |
| scf | – | – | 0 | – | 0 | 1 | Carry wird gesetzt |
| ccf | – | – | ? | – | 0 | ! | Carry-Komplement |
| bit b,s | ? | ! | 1 | ? | 0 | – | Test von Bit b |

Zeichenerklärung:
! Flag wird entsprechend beeinflußt
? Flag wird willkürlich beeinflußt
1 Flag wird gesetzt
0 Flag wird zurückgesetzt
– Flag wird nicht beeinflußt

**Übersicht der Flags:** Die aufgelisteten Befehle wirken gemäß der Zeichenerklärung auf die sechs Flags. Wichtig sind das Carry- und das Zero-Flag

Eine andere Möglichkeit Befehle einzuordnen, bietet die Adressierung. Die Adressierungsart ist immer von den Operanden (in der Assemblerschreibweise) abhängig. Damit nun etwas Ordnung in Begriffe wie „indirekter Ladebefehl" oder „absoluter Sprung" kommt, betrachten wir die einzelnen Arten der Adressierung einmal namentlich.

○ Eine *direkte* oder *unmittelbare* Adressierung liegt immer dann vor, wenn ein Befehl einen 8-Bit- oder 16-Bit-*Zahlenwert* enthält. Der 8-Bit- oder 16-Bit-Wert erscheint dann immer hinter dem Operationscode. Direkte Adressierung gibt es bei Lade-, Arithmetik- und bei Logikbefehlen.

Beispiele: *ld a,9C*
*ld bc,FFFF*
*add a,1C*
*cp FF*

○ *Absolut* heißt eine Adressierungsart, wenn eine 16-Bit-*Adresse* den Operanden darstellt. Die beiden Adreßbytes sind wieder im Befehl selbst enthalten und erscheinen in umgekehrter Reihenfolge hinter dem Operationscode. Ladebefehle und Sprünge sind absolut adressierbar.

Beispiele: *ld a,(4026)*
*ld bc,(400C)*
*call 4082*

○ Bei der *Register*-Adressierung werden ausschließlich Register verwendet. Die Register-Adressierung ist fast überall zu finden.

Beispiele: *ld c,l*
*adc a,b*
*cp b*

○ Eine *indirekte* Adressierung liegt dann vor, wenn ein Registerpaar die *Adresse* des Operanden enthält. Die indirekte Adressierung ist vornehmlich bei Ladebefehlen zu finden.

Beispiele: *ld a,(bc)*
*jp (hl)*

○ Die *relative* Adressierung ist nur bei Sprüngen möglich. Dabei legt ein 8-Bit-Zahlenwert die Zieladresse in bezug auf den gegenwärtigen Programmzählerstand fest.

Beispiel: *jr FE*

○ *Zero-Page*-Adressierung ist nur bei Restarts möglich. Dabei ist die höherwertige Seite der 16-Bit-Zieladresse automatisch Null.

Beispiel: *rst 10*

○ Die *bitweise* Adressierung ermöglicht die Einflußnahme auf einzelne Bits im Operanden. Diese Adressierung gibt es vornehmlich bei Einzelbitbefehlen.

Beispiele: *set 3,(hl)*
*res 2,a*
*bit 6,c*

Eine allgemeingültige Klassifizierung der Befehle nach diesen Kriterien ist indes kaum möglich! Sobald nämlich ein zweiter Operand ins Spiel kommt, wird die Angelegenheit kompliziert. Ist nun der Befehl *ld (hl),N* zur direkten oder zur indirekten Adressierung zu zählen? Eine hieb- und stichfeste Antwort läßt sich hier nicht geben.

Fazit: Adressierungsarten eignen sich nur bedingt zum Einordnen von Befehlen; sie eignen sich aber sehr gut, um zu beschreiben, wie die Zentraleinheit (CPU) die einzelnen Operanden verarbeitet.

Klaus Herklotz

(*Wird fortgesetzt*)

## ZX-81-Hardwaretip:

# Störungsfreies Fernsehbild

Bei einigen ZX-81-Computern – besonders bei solchen, die innerhalb des letzten halben Jahres auf den Markt kamen – kommt es beim Anschluß von Peripheriegeräten (z. B. 16 KByte RAM) zu Störungen auf dem Bildschirm. Der Grund: Durch eine Modifikation auf der Platine kommt der Spannungsregler des ZX 81 ins Schwingen. Mehr oder weniger starke Schlieren am Bildschirm sind das Ergebnis. Abhilfe schafft ein Abblock-Kondensator (ca. 68 µF/16 V), den man, wie im *Bild* gezeigt, an den Spannungsregler anlötet. –li



**ZX-81-Platine:** Ein zusätzlicher Abblock-Kondensator am Spannungsregler verhindert Bildstörungen

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 20: Datentransport im Paket

**Blocktransferbefehle zum Kopieren ganzer Datenblöcke sind so ziemlich das Leistungsstärkste, was die Z-80-CPU zu bieten hat.**

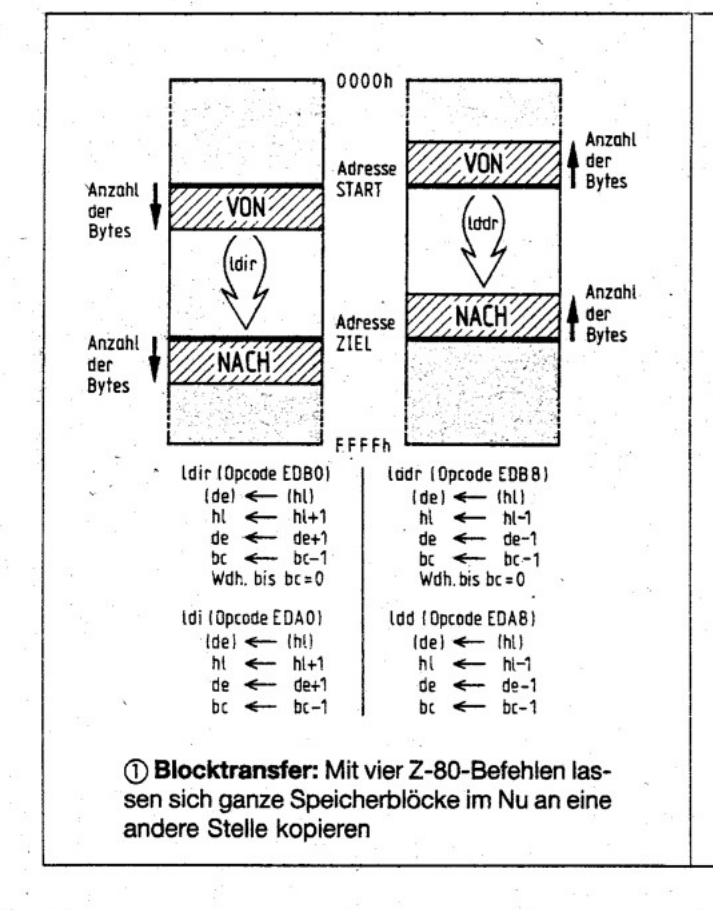
Ein Blocktransfer kopiert eine gewünschte Anzahl benachbarter Bytes von einem Teil des Speichers (START) in einen anderen (ZIEL). Bevor der Blocktransfer ausgeführt wird, müssen lediglich drei Registerpaare mit den jeweiligen Anfangsdaten geladen werden (*Bild* 1):

Registerpaar bc erhält die Anzahl der Bytes, die kopiert werden sollen, und dem hl-Registerpaar wird die Startadresse des ersten zu verschiebenden Bytes zugewiesen. Die Zieladresse dieses ersten Bytes erhält das de-Registerpaar.

Abschließend muß man noch entscheiden, ob der Transfer die Bytes oberhalb oder unterhalb der Startadresse betrifft: *ldir* überträgt die gewünschte Anzahl von Bytes unterhalb der Startadresse, während *lddr* die Bytes oberhalb der Startadresse überträgt.

Blocktransferbefehle können also als Schleife mit dem 16-Bit-Zählregister bc aufgefaßt werden (Bild 1): Innerhalb dieser Schleife wird der Inhalt der Speicherzelle, die das hl-Registerpaar festlegt, in die Speicherzelle kopiert, die das de-Registerpaar festlegt. Danach werden beim *ldir*-Befehl die Inhalte der Registerpaare de und hl um 1 erhöht, beim *lddr*-Befehl um 1 verringert. Den Abschluß der Schleife bilden zwei Befehle: Der eine verringert den Stand des Schleifenzählers bc um 1; der andere bewirkt einen Sprung zum Anfang der Schleife, solange der Stand des Schleifenzählers noch nicht 0 ist.

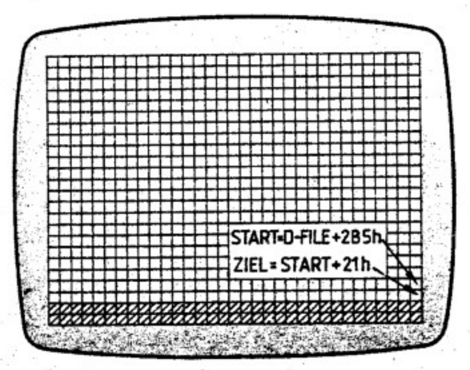
Geringfügig anders wird ein Blocktransfer mit den Befehlen *ldi* und *ldd* vollzogen. Hierbei fehlt lediglich der Sprung am Schluß der Schleife, d. h. der Programmierer kann noch einige Bytes in die Transferschleife einfügen. Zu beachten ist dann, daß das P/V-Flag eine Sonderaufgabe erfüllt: Es zeigt an, ob der Stand des Schleifenzählers schon 0 ist. Solange dies nicht der Fall ist, bleibt auch das P/V-Flag auf logisch 1. Die Transferschleife muß deshalb mit dem Befehl *jp pe, NN* abgeschlossen werden!




① **Blocktransfer:** Mit vier Z-80-Befehlen lassen sich ganze Speicherblöcke im Nu an eine andere Stelle kopieren

| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | 2A | 0C | 40 | | | ld hl,(400C) |
| 4085 | E5 | | | | | push hl |
| 4086 | 01 | B5 | 02 | | | ld bc,02B5 |
| 4089 | 09 | | | | | add hi,bc |
| 408A | E5 | | | | | push hl |
| 408B | 11 | 21 | 00 | | | ld de,0021 |
| 408E | 19 | | | | | add hl,de |
| 408F | EB | | | | | ex de,hl |
| 4090 | E1 | | | | | pop hl |
| 4091 | ED | B8 | | | | lddr |
| 4093 | E1 | | | | | pop hl |
| 4094 | 06 | 20 | | | | ld b,20 |
| 4096 | 23 | | | | CLEAR | inc hl |
| 4097 | 36 | 00 | | | | ld (hl),00 |
| 4099 | 10 | FB | | | | djnz CLEAR |
| 409B | C9 | | | | | ret |

② **Pseudo-SCROLL:** Jeder Aufruf dieses Programms bewegt den Bildinhalt um eine Zeile nach unten




③ **Zeilentransfer:** Der Inhalt der Bildspeicherzelle rechts unten wird als erstes verschoben

## Ein neuer SCROLL-Befehl

Im Basic des ZX 81 gibt es den Befehl SCROLL, der den Bildschirminhalt um eine Zeile nach oben bewegt. Die Maschinenroutine in *Bild 2* bewegt den Bildschirminhalt dagegen um eine Zeile nach unten. Die Bytes dieser Routine können wie gewohnt in eine REM-Zeile untergebracht werden (Eingabeprogramm von Teil 3). SCROLL-Befehle eines Basic-Programms ersetzt man nun am besten durch:

```
LET Q = USR 16514
```

Im Gegensatz zur ursprünglichen SCROLL-Routine wird die PRINT-Position dann aber nicht verändert! Und so kommt es dazu:

Zuerst hilft wie üblich die Systemvariable D-FILE beim Laden des hl-Registerpaares mit der Adresse der Bildspeichergrenze (4082h). Dieser Wert ist auch nach dem Transfer noch von Bedeutung und wird deshalb auf den Stapel geworfen (4085h). Das Registerpaar bc erhält anschließend die Zahl der zu verschiebenden Bytes (4086h). Wie aus der Bildschirmskizze (*Bild 3*) hervorgeht, ist dieser Wert 2B5h (dezimal 693 = 21 Zeilen × 33 Spalten) zugleich der Summand, der zu D-FILE im hl-Registerpaar addiert, die Adresse START ergibt (4089h). Um diesen Wert zu retten, wirft *push hl* eine Kopie auf den Stapel (408Ah).

Registerpaar de muß die Zieladresse erhalten. Dazu ist bloß 21h (dezimal 33; Länge einer Bildschirmzeile) zur Startadresse im hl-Registerpaar zu addieren (408Bh bis 408Eh) und das Ergebnis ins de-Registerpaar zu kopieren (408Fh). Anschließend erhält das Registerpaar hl wieder die Adresse START (4090h), und *lddr* führt den Transfer aus (4091h). Der Programmteil nach dem Blocktransfer dient lediglich dazu, die oberste Bildschirmzeile mit Leerzeichen zu überschreiben.

## Jetzt wird der Bildspeicher beweglich

Nun wollen wir den Transfer des gesamten Bildspeicherinhalts in einen anderen Speicherbereich in Angriff nehmen. Dazu wären 725 Speicherzellen einer REM-Zeile durch Eintippen ebensovieler Zeichen für den Bildschirminhalt zu reservieren. Das gibt wunde Finger! Aber wozu haben wir RAMTOP!?

Die Systemvariable RAMTOP enthält die Adresse der ersten Speicherzelle, die für Basic-Programme nicht erreichbar ist: Alles was über RAMTOP liegt, ist deshalb auch vor NEW geschützt. Nach dem Einschalten des ZX 81 liegt RAMTOP immer am Ende des RAM-Bereichs (auch mit 16 KByte RAM). Wenn wir nun den Wert in RAMTOP verringern, ist oberhalb des neuen Wertes Speicherplatz z. B. für den Bildinhalt reserviert. Ein Bildinhalt belegt 725 Speicherzellen (einschließlich der NEWLINE-Codes; siehe Teil 2). Das folgende Programm setzt RAMTOP sicherheitshalber etwas tiefer:

```
10 LET A=PEEK 16388+
   256*PEEK 16389
20 LET A=A-750
30 POKE 16388,A-256*INT (A/256)
40 POKE 16389,INT (A/256)
50 NEW
```

Das Programm schafft nach RUN den erwünschten Speicherplatz und löscht das Hilfsprogramm, wobei erst NEW die RAMTOP-Verschiebung wirksam macht. Man darf anschließend von Band ein Eingabeprogramm für das Transferprogramm (*Bild 4*) laden.

Mittel zum Zweck ist wieder ein Blocktransfer. Das bc-Registerpaar erhält die Anzahl der zu speichernden Bytes (2D6h), das hl-Registerpaar die Startadresse (Bildspeichergrenze D-FILE) und das de-Registerpaar die Zieladresse mit Hilfe von RAMTOP. Der *nop*-Befehl hält eine Speicherzelle zur späteren Verwendung frei, den Blocktransfer führt diesmal *ldir* durch.

Wie holen wir aber den abgespeicherten Bildschirminhalt wieder auf den Bildschirm? Hier hilft ein kleiner Kunstgriff, genannt Mittenprogramm-Änderung. Damit der Transfervorgang umgekehrt abläuft, brauchen bloß die Start- und Zieladressen vertauscht zu werden: POKE 16524,235 ersetzt das *nop*-Byte bei Adresse 408Ch (16524) durch den Befehl *exde,hl*. Mit dem Maschinenprogramm-Aufruf RAND USR 16514 kommt nun das abgespeicherte Bild auf den Bildschirm. Folgendes Demonstrationsprogramm verdeutlicht das:

*Speichern*

```
200 FOR N=1 TO 70
210 PRINT "FUNKSCHAU";
220 NEXT N
230 POKE 16524,0
240 RAND USR 16514
```

*Abrufen*

```
POKE 16524,235
RAND USR 16514
```

Nach dem Abrufen wird der Bildschirm blitzartig mit dem Text „FUNKSCHAU" vollgeschrieben. Erweitert man das Programm so, daß mehrere Bilder auf diese Weise abzurufen sind, dann ist z. B. eine Art Zeichentrick-Darstellung mit beweglichen Figuren möglich.

Klaus Herklotz

*(Wird fortgesetzt)*

④ **Bildtransfer:** Das Programm kopiert den Bildspeicher in einen Bereich oberhalb von RAMTOP

| Adresse | | | | | Befehl |
|---|---|---|---|---|---|
| 4082 | 01 | D6 | 02 | | ld bc,02D6 |
| 4085 | 2A | 0C | 40 | | ld hl,(400C) |
| 4088 | ED | 5B | 04 | 40 | ld de,(4004) |
| 408C | 00 | | | | nop |
| 408D | ED | B0 | | | ldir |
| 408F | C9 | | | | ret |

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 21: Auf der Suche nach Byte X

**Flugs zum Ziel führen „Suchbefehle“ des Z-80-A-Mikroprozessors, wenn man den Speicher des ZX 81 nach ganz bestimmten Bytes abklappern möchte.**

Suchbefehle sind ein sehr wirkungsvolles Mittel, um z. B. Programmänderungen schnell auszuführen. Dazu ein Beispiel: In einem Programm sollen alle PRINT-Befehle durch LPRINT ersetzt werden oder die Variable Z ist in X umzubenennen. Dies von Hand auszuführen, wäre bei langen Programmen sehr mühsam. Ein Maschinenprogramm mit Suchbefehlen zum Aufspüren der Codes von PRINT bzw. Z hilft dagegen die Aufgabe schnell und elegant zu lösen.

### Suchbefehle sind komplette Miniprogramme

Der Umgang mit Suchbefehlen ähnelt stark dem mit Blocktransferbefehlen. Denn wieder sind als erstes die Anfangsdaten in Registern unterzubringen.

Registerpaar bc erhält die Anzahl der Speicherzellen zugewiesen, die abzusuchen sind. Die Adresse der ersten abzusuchenden Speicherzelle wird ins hl-Registerpaar geladen, den hexadezimalen Code des zu suchenden Bytes erhält der Akkumulator.

Abschließend muß man entscheiden, in welche Richtung die Suche gehen soll: *cpir* sucht unterhalb der Startadresse, während *cpdr* oberhalb der Startadresse sucht (*Bild 1*).

Suchbefehle sind als Schleife mit dem 16-Bit-Zähler bc aufzufassen. Innerhalb der Schleife vergleicht ein compare-Befehl den Akkumulatorinhalt mit dem Inhalt der Speicherzelle, die das hl-Registerpaar festlegt. Sollten beide Bytes identisch sein, dann wird das Zero-Flag gesetzt.

Nachfolgend erhöht der *cpir*-Befehl das hl-Registerpaar um 1, während der *cpdr*-Befehl es um 1 vermindert. Beide Befehle reduzieren danach auch den Schleifenzähler bc um 1. Solange der Stand dieses Schleifenzählers nicht 0 wurde, bleibt auch das P/V-Flag gesetzt. Abschließend erfolgt ein Sprung zum Anfang der Schleife, solange der Schleifenzähler nicht auf 0 steht *und* das entsprechende Byte nicht gefunden wurde.

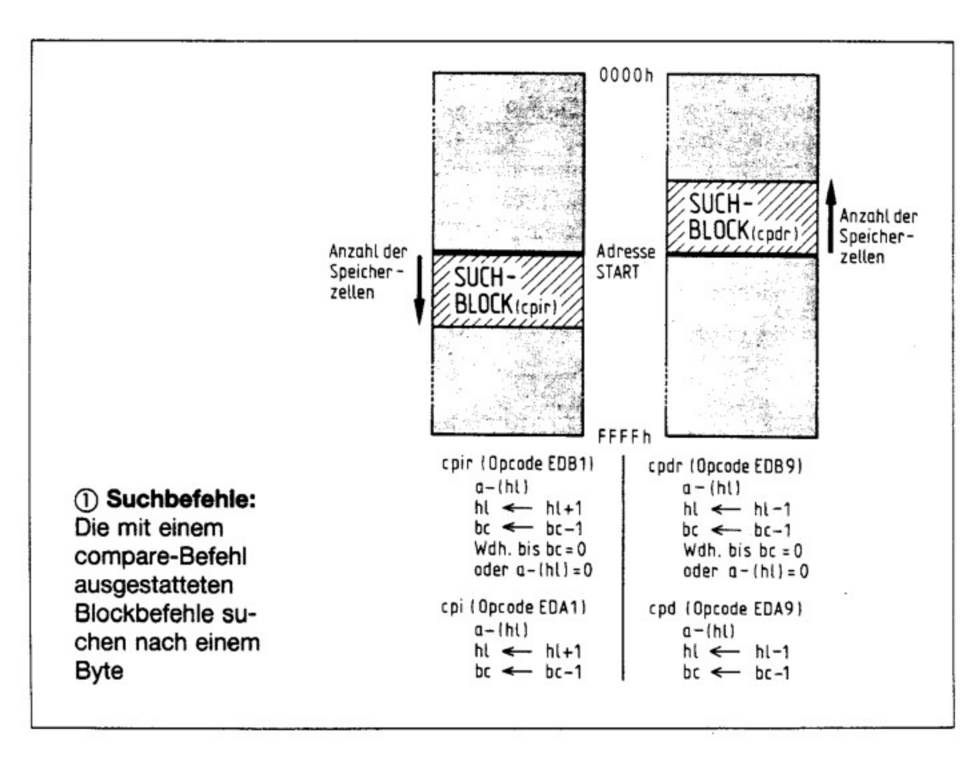
Wurde dagegen die Schleife verlassen, dann kann man die Ursachen dafür an den Flags ablesen: Wenn das P/V-Flag zurückgesetzt ist, wurde die Schleife vollständig abgearbeitet. Sollte das Zero-Flag gesetzt sein, so wurde das Byte gefunden. Seine Adresse befindet sich im hl-Registerpaar.

Zusätzlich zu diesen beiden abgeschlossenen Suchbefehlen gibt es wieder „offene“ Abwandlungen: Bei *cpi* und *cpd* fehlt lediglich der Sprung zum Anfang der Schleife. Der Programmierer kann somit wieder einige Bytes einfügen.

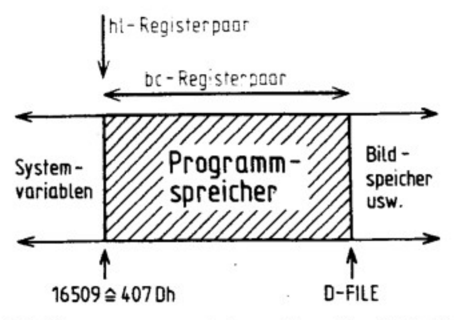
### STOP-Befehle im Visier

Jedes selbstgeschriebene Basic-Programm hat am Anfang seine Fehler. Bei der systematischen Fehlersuche sollten dann Programmteile einzeln getestet werden. Diese Programmteile gegeneinander abzugrenzen ist Aufgabe von eingefügten STOP-Befehlen. Sind später alle Fehler aufgespürt und beseitigt, ist es oft ein mühsames Unterfangen, alle STOP-Befehle wieder zu löschen.

Für diese Aufgabe können wir jetzt Suchbefehle verwenden: Sie sollen den Programmspeicher nach unerwünschten STOP-Zeilen absuchen. Sind diese Zeilen erst einmal gefunden, so kann ein




① **Suchbefehle:** Die mit einem compare-Befehl ausgestatteten Blockbefehle suchen nach einem Byte




② **Programmspeicher:** Den Suchblock grenzen die absolute Adresse 16509 und die indirekte Adresse D-FILE ein

einfacher Ladevorgang sie in harmlose REM-Zeilen verwandeln.

Zuerst müssen die Grenzen des Suchbereichs ermittelt werden (*Bild 2*). Die untere Grenze davon liegt fest auf 16509 (Beginn des Programmspeichers), die obere ist in der Systemvariablen D-FILE versteckt. Registerpaar hl kann somit den absoluten Wert 407Dh erhalten, das bc-Registerpaar muß dagegen mit der Differenz aus D-FILE und 16509 – also dem Umfang des Suchbereichs – geladen werden. Diese Differenz zu bilden ist mitunter Aufgabe des Maschinenprogramms aus *Bild 3*.

Zuerst wird dem hl-Registerpaar mit Hilfe von D-FILE die Adresse der Obergrenze des Suchbereichs zugewiesen (4082h) und das de-Registerpaar erhält die Adresse der Untergrenze (4085h). Dann setzt der Befehl *and a* sicherheitshalber das Carry-Flag zurück, damit der nachfolgende Befehl keinen Übertrag berücksichtigt: *sbc hl,de* bildet die gewünschte Differenz.

Vor dem Ausführen des Suchbefehls muß das hl-Registerpaar die untere Grenze des Suchbereichs erhalten. Da diese Adresse aber noch im de-Registerpaar steht, ist lediglich ein Austausch vorzunehmen (408Bh). Danach befindet sich die Differenz (Umfang des Suchbereichs) im de-Registerpaar. Durch einfachen Datentransport (408Ch und 408Dh) wird dieser Wert ins bc-Registerpaar gebracht (leider gibt es keinen Befehl *ex de,bc*). Als letzte Station vor dem Suchbefehl erhält der Akku den Code des Suchbytes.

Die folgende Suchschleife *cpir* wird nur dann verlassen, wenn entweder der gesamte Bereich abgesucht (P/V = 0), oder ein STOP-Befehl aufgespürt wurde. Im ersten Fall beendet ein Rücksprung ins Basic die Routine (4092h), ansonsten wird der STOP-Befehl in eine REM-Anweisung umgewandelt (4094h) und weitergesucht (4097h).

## Suchbefehle sind ungenau

Wer mit Suchbefehlen arbeitet, ist zu besonderer Wachsamkeit verpflichtet. Ist nämlich ein Suchbyte gefunden, so werden dennoch die In- und Dekrementierungen noch einmal durchgeführt (Bild 1). Um die wahre Adresse des Suchbytes zu erhalten, muß daher das hl-Registerpaar um 1 vermindert werden! Dies erklärt denn auch die beiden Arithmetikbefehle *dec hl* (4093h) und *inc hl* (4096h).

Für einen ersten Test kann unser Eingabeprogramm aus Teil 3 verwendet werden, da es selbst zwei STOP-Befehle erhält. Nach dem POKE-Vorgang wandelt

RAND USR 16514

die STOP-Zeilen in REM-Zeilen um. Aber Vorsicht! Nach jedem Umwandlungsvorgang muß das Eingabeprogramm neu gestartet werden – denn auch der Inhalt der Speicherzelle 408Fh wurde verändert.

Und noch eine Tücke ist zu beachten: Taucht in unserem Fall die Zeilennummer 227 in einem Basic-Programm auf, dann macht das Maschinenprogramm daraus kurzerhand die Zeilennummer 234 (Dezimalcode von REM). Man sollte deshalb stets vor Augen haben, daß im Suchbereich jedes mit dem Suchbyte identische Byte einen anderen Wert bekommt. Wollte man z. B. die Variable A durch B ersetzen und taucht auch in REM- oder PRINT-Anweisungen der Buchstabe A auf, so wird er dort ebenfalls durch B ersetzt.

Wie könnte man das Programm aus Bild 3 nun ergänzen, damit wirklich nur STOP-Zeilen manipuliert werden? Ein Tip: Das Byte nach jedem STOP-Befehl trägt den NEW-LINE-Code (118)!

Klaus Herklotz

(*Wird fortgesetzt*)

| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | 2A | 0C | 40 | | | ld hl,(400C) |
| 4085 | 11 | 7D | 40 | | | ld de,407D |
| 4088 | A7 | | | | | and a |
| 4089 | ED | 52 | | | | sbc hl,de |
| 408B | EB | | | | | ex de,hl |
| 408C | 4B | | | | | ld c,e |
| 408D | 42 | | | | | ld b,d |
| 408E | 3E | E3 | | | | ld a,E3 |
| 4090 | ED | B1 | | | WEITER | cpir |
| 4092 | E0 | | | | | ret po |
| 4093 | 2B | | | | | dec hl |
| 4094 | 36 | EA | | | | ld (hl),EA |
| 4096 | 23 | | | | | inc hl |
| 4097 | 18 | F7 | | | | jr WEITER |

③ **Anti-STOP-Programm:** Sämtliche STOP-Befehle eines Basic-Programms werden von diesem Maschinenprogramm durch REM-Befehle ersetzt

## Klartext im Detail:

# Nomenklatur

In Teil 20 der Serie „Klartext für den ZX 81" geht es um die Blocktransferbefehle der Z-80-CPU. Recht schleierhaft erscheinen dabei die Mnemonics *ldir*, *lddr*, *ldi* und *ldd*. Warum werden die Blocktransferbefehle gerade so abgekürzt?

Das *ld* in jedem Kürzel bedeutet load (laden) und weist auf den einfachen Datentransport hin. Mit *i* ist increment (um 1 vergrößern), mit *d* decrement (um 1 verringern) gemeint. Das r am Schluß steht schließlich für repeat (Wiederholung). Klaus Herklotz

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 22: Dez statt Hex

**Schon ein einziger Z-80-Befehl genügt, und wir müssen einen Akkuinhalt nicht mehr hexadezimal interpretieren, sondern dürfen ihn dezimal auffassen.**

Um es gleich vorwegzunehmen: Diesmal machen wir aus dem ZX 81 mit einem Programm einen Ereigniszähler. Die nötigen Gedankenschritte sind am Flußdiagramm (*Bild 1*) leicht nachvollziehbar.

Im Rahmen der Vorbereitungen muß erst einmal ein Zählregister zurückgesetzt werden. Sobald nun eine Taste gedrückt wird, erhöht das den Inhalt eben dieses Zählregisters um 1 und der Inhalt wird ausgegeben. Weil dieser Vorgang sehr schnell geht, muß der Computer darauf warten, daß die Taste wieder losgelassen wird. Erst danach darf sich der Vorgang wiederholen.

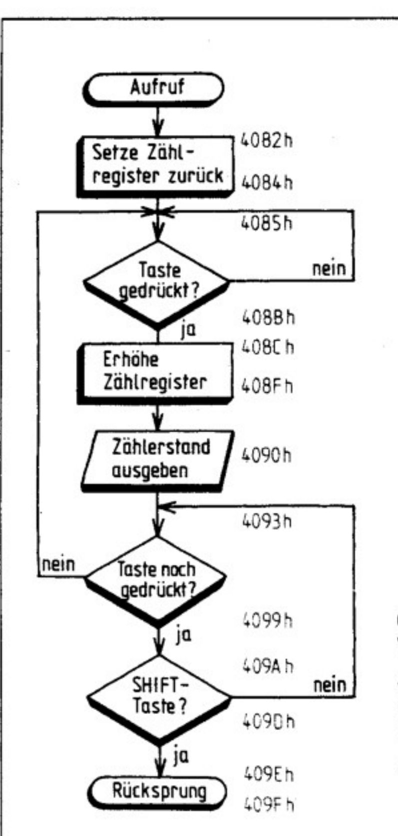
### Der Stapel rettet das Zählregister

Das fertige Programm (*Bild 2*) sorgt zuerst fürs Rücksetzen und Retten des Akkumulatorinhalts vor der ersten Tastaturabfrage. Wegen der folgenden Schleife (Adressen 4085h bis 408Bh) wartet der Computer dann so lange, bis eine Taste gedrückt wird (siehe Teil 11). Erst wenn dies geschehen ist, beginnt der eigentliche Zählvorgang. Dabei holt sich der Computer den Akkumulatorinhalt vom Stapel zurück (408Ch). *inc a* erhöht anschließend den Akkumulatorinhalt um 1 und ein nop-Byte reserviert eine Speicherzelle zur späteren Verwendung.

Nach dem erneuten Retten des Akkuinhalts (408Fh) gibt das schon bekannte

**② Zählprogramm:** Für die Ausgabe des Zählerstandes sorgt das altbekannte Unterprogramm WRITE

| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | 3E | 00 | | | | ld a,00 |
| 4084 | F5 | | | | | push af |
| 4085 | 3A | 26 | 40 | | START | ld a,(4026) |
| 4088 | FE | FF | | | | cp FF |
| 408A | 28 | F9 | | | | jr z,START |
| 408C | F1 | | | | | pop af |
| 408D | 3C | | | | | inc a |
| 408E | 00 | | | | | nop |
| 408F | F5 | | | | | push af |
| 4090 | CD | A0 | 40 | | | call WRITE |
| 4093 | 3A | 26 | 40 | | WAIT | ld a,(4026) |
| 4096 | FE | FF | | | | cp FF |
| 4098 | 28 | EB | | | | jr z,START |
| 409A | FE | FE | | | | cp FE |
| 409C | 20 | F5 | | | | jr nz,WAIT |
| 409E | F1 | | | | | pop af |
| 409F | C9 | | | | | ret |
| | | | | | | |
| 40A0 | ED | 5B | 0C | 40 | WRITE | ld de,(400C) |
| 40A4 | 13 | | | | | inc de |
| 40A5 | F5 | | | | | push af |
| 40A6 | CB | 3F | | | | srl a |
| 40A8 | CB | 3F | | | | srl a |
| 40AA | CB | 3F | | | | srl a |
| 40AC | CB | 3F | | | | srl a |
| 40AE | C6 | 1C | | | | add a,1C |
| 40B0 | 12 | | | | | ld (de),a |
| 40B1 | 13 | | | | | inc de |
| 40B2 | F1 | | | | | pop af |
| 40B3 | E6 | 0F | | | | and 0F |
| 40B5 | C6 | 1C | | | | add a,1C |
| 40B7 | 12 | | | | | ld (de),a |
| 40B8 | C9 | | | | | ret |




**① Flußdiagramm:** Wenn wir den ZX 81 zum Zähler machen möchten, sind diese Programmschritte zweckmäßig

Unterprogramm WRITE den Akkuinhalt aus (4090h). Aufgrund den folgenden Programmschritte wartet der Computer nun in einer Schleife, bis die gewiß noch gedrückte Taste wieder losgelassen wird (4098h bis 409Dh). Nur wenn die SHIFT-Taste gedrückt war, erfolgt sofort ein Rücksprung ins Basic (409Fh).

Die Bytes des Programmes bringen wir wie gewöhnlich in eine REM-Zeile unter. Zum Starten genügt danach die Eingabe von

RAND USR 16514

Wenn man jetzt eine Taste niederdrückt, wird der Zählerstand sichtbar. Die SHIFT-Taste beendet den Zählvorgang, wogegen jede andere Taste den Zählerstand um 1 erhöht. Ist dann eine Taste z. B. 33mal gedrückt worden, meldet das Zählprogramm die Zahl 21 (hexadezimal). Etwas ärgerlich ist dies schon, denn auch noch so große Computer-Fanatiker könnten mit der zugehörigen Dezimalzahl bestimmt mehr anfangen. Unsere nächste Aufgabe ist es daher, den Computer aufs Dezimalsystem zu trimmen.

Ein neuer Befehl nimmt uns dabei mit einer Rechenoperation die Arbeit ab. Sobald nämlich eine der beiden Hex-Ziffern im Akku einen Wert annimmt, der im Dezimalsystem nicht definiert ist (Ziffern A bis F), muß diese Ziffer mit Rücksicht auf einen Übertrag korrigiert werden: Ist also eine Ziffer im Akku größer als 9, soll einfach 6 dazugezählt werden.

Machen wir uns das gleich am Zählprogramm klar: POKE 16526,39 füllt die reservierte Speicherzelle 408Eh mit dem dezimalen Befehlscode des Korrekturbefehls *daa* (Decimal Adjust Accu: Dezimalabgleich im Akku). Nach dem Programmstart mit

RAND USR 16514

bringen wir dann durch wiederholtes Tastendrücken die Zahl 09 auf den Bildschirm. Wenn jetzt nochmals eine Taste gedrückt wird, erhöht das den Akkumulatorinhalt zwangsläufig auf 0Ah.

Der nachfolgende daa-Befehl „merkt" aber, daß die rechte Ziffer im Akku größer als 9 ist und addiert selbsttätig 6 dazu. Im Akku und am Bildschirm steht jetzt die Zahl 10 (9h + 6h = 10h), die wir zwar als Dezimalzahl auffassen, die für den ZX 81 aber weiterhin eine Hexadezimalzahl bleibt!

FALL 1: Akku 2, b-Register 1 → Akku 3
Soll-Ergebnis: 28 + 11 = 39
⟹ Kein Abgleich nötig

FALL 2: Akku 2, b-Register 1 → Akku 3
Soll-Ergebnis: 28 + 13 = 41
⟹ Abgleich nötig: 3Bh + 6h = 41h

FALL 3: Akku 2, b-Register 1 → Akku 4
Soll-Ergebnis: 28 + 19 = 47
⟹ Abgleich nötig: 41h + 6h = 47h

FALL 4: Akku 3, b-Register 1 → Akku 2
Soll-Ergebnis: 31 - 11 = 20
⟹ Kein Abgleich nötig

FALL 5: Akku 3, b-Register 1 → Akku 1
Soll-Ergebnis: 31 - 13 = 18
⟹ Abgleich nötig: 1Eh - 6h = 18h

FALL 6: Akku 3, b-Register 1 → Akku 1
Soll-Ergebnis: 31 - 19 = 12
⟹ Abgleich nötig: 18h - 6h = 12h

③ **Planspiel:** Ob bei den sechs Arithmetik-Opererationen ein Dezimalabgleich stattfindet, hängt vom tatsächlichen und erwünschten Ergebnis ab

## Flags leisten Spitzeldienste

Wie reagiert der Dezimalabgleich aber auf eine Subtraktion, wie sie etwa der Befehl *dec a* ausführt? Am besten probieren wir das aus. Nach Drücken der SHIFT-Taste (Programmabbruch) bringt POKE 16525,61 den *dec-a*-Befehlscode in die Speicherzelle 408Dh und POKE 16515,154 läßt den Zählvorgang gleich bei 99 beginnen. Nach

RAND USR 16514

und wiederholten Tastendrücken wird bald die Zahl 90 am Bildschirm stehen. Der nächste Tastendruck vermindert jetzt den Akkumulatorinhalt auf den Wert 8Fh. Der Dezimalabgleich erkennt dies wieder, zieht diesmal aber 6 ab! Am Bildschirm steht nun die Zahl 89 (8Fh – 6h = 89h).

Eine grundlegende Eigenschaft des Dezimalabgleichs *daa* steht damit fest: Er reagiert nach einer Subtraktion anders als nach einer Addition. Um das zu gewährleisten, stellt die CPU mit Hilfe des N-Flags (siehe Teil 19) fest, welcher Art die letzte arithmetische Operation war. Deshalb setzen auch prinzipiell alle Subtraktionsbefehle das N-Flag, wogegen es Additionsbefehle zurücksetzen. Zwischen dem *daa*-Befehl und der „abzugleichenden" Operation darf daher kein Befehl stehen, der das N-Flag beeinflußt. Nur dann ist ein korrekter Abgleich gewährleistet.

Bis jetzt haben wir nur ins Auge gefaßt, wie der Dezimalabgleich nach Inkrement- und Dekrementbefehlen funktioniert. Ein Planspiel für allgemeine Arithmetikoperationen (*Bild 3*) klärt die übrigen Fälle.

Bei Fall 3 und Fall 6 tritt etwas Merkwürdiges auf: Obwohl keine verbotene Ziffer (A bis F) im Ergebnis vorliegt, muß abgeglichen werden! Auch dazu erhält die CPU Unterstützung von den Flags. In beiden Fällen liegt nämlich ein Übertrag vom vierten Bit zum fünften Bit bzw. umgekehrt vor.

Konsequenterweise hat die CPU dann das H-Flag gesetzt. Und immer wenn das H-Flag gesetzt ist, wird vom *daa*-Befehl die rechte Ziffer korrigiert, sobald das C-Flag gesetzt ist, muß er die linke Ziffer korrigieren. Prompt kommt unser Zähler auch außer Tritt, wenn wir über 0 hinaus abwärtszählen, denn dabei wird das C-Flag gesetzt und von der linken Ziffer unbeabsichtigt 6 abgezogen.

Klaus Herklotz

(Wird fortgesetzt)

Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 23: Noch zwei Registerpaare!

**Diesmal lernen wir zwei neue Registerpaare kennen, die allerdings etwas aus der Reihe tanzen. Die zugehörigen Operationscodes setzen der CPU deshalb erst einmal einen Warnschuß vor den Bug.**

In Anbetracht des bisher kennengelernten Z-80-Befehlssatzes müssen wir dem hl-Registerpaar eine Sonderstellung gegenüber den anderen beiden Registerpaaren zubilligen. So erlaubt das hl-Registerpaar vielseitige Operationen auch im 8-Bit-Bereich: Einzelbit-. Arithmetik-, Logik- sowie Rotations- und Schiebeoperationen sind durchs hl-Registerpaar indirekt adressierbar.

### Zwei Verwandte des hl-Registerpaares

Es gibt jedoch noch zwei weitere Registerpaare, die einen ähnlichen Datentransport wie das hl-Registerpaar zulassen. Gemeint sind die beiden 16 Bit breiten Indexregister x und y. Die Kürzel in der Assemblerschreibweise lauten ix und iy.

Ihre Ähnlichkeit mit dem hl-Registerpaar macht insbesondere der 16-Bit-Befehlsvorrat (*Bild*) deutlich: Die Operationscodes dieser 16-Bit-Operationen sind mit denen des hl-Registerpaares fast identisch. Beim ix-Register steht lediglich DDh, beim iy-Register ein Byte mit dem Wert FDh voran. Diese beiden „Vorsilben" gebieten dem Z-80-Prozessor: Vorsicht! Die nachfolgende Operation darf nicht das hl-Registerpaar, sondern muß eines der Indexregister durchführen.

Das Zusammenfassen einzelner Register zu Registerpaaren ermöglichte mitunter die indirekte Adressierung beim Datentransport. Nur für die beiden Indexregister kommt jetzt noch die „indizierte" Adressierung dazu. Wie sie funktioniert, ist schnell erklärt.

### Ein Nachschlag für die Indexregister

Indizierte Adressierung ist prinzipiell bei allen 8-Bit-Operationen möglich. Um diese Adressierung ausführen zu können, erhält der Operationscode ein zusätzliches Datenbyte, ähnlich dem beim relativen Sprung. Der Prozessor addiert dieses Datenbyte komplementär zum Inhalt des betroffenen Indexregisters. Das Ergebnis ist dann die endgültige Adresse.

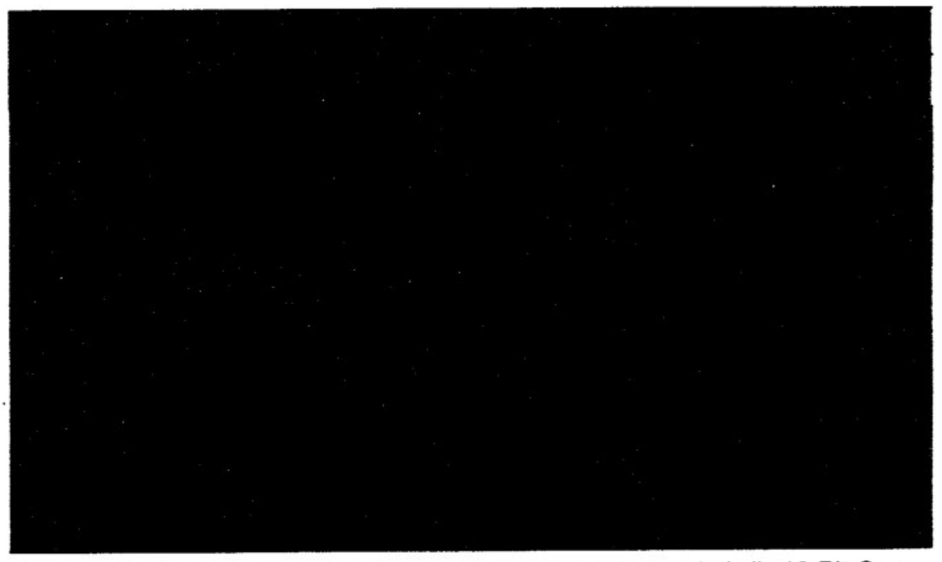
Dazu ein Beispiel: Im iy-Register soll bereits die Zahl 4000h stehen. Der Befehl *ld a, (iy+26)* addiert 26h zum Indexregisterinhalt 4000h und ergibt so die absolute Adresse 4026h, deren Inhalt in den Akku geladen wird. Im Gegensatz dazu subtrahiert der Ladebefehl *ld a, (iy+FA)* die Zahl 06h vom Indexregisterinhalt 4000h und lädt deshalb den Inhalt der endgültigen Adresse 3FFAh in den Akku.

Hier läßt sich eine weitere Parallele zum relativen Sprung erkennen: Sollte der Wert des Datenbytes zwischen 00 und 7Fh liegen, so wird die endgültige Adresse durch Addieren gewonnen. Ist der Wert des Datenbytes dagegen größer, dann wird das Zweierkomplement (siehe auch Teil 12) dieser Zahl subtrahiert.

Eine Liste aller erlaubten 8-Bit-Operationen mit den Indexregistern folgt im nächsten Teil, ebenso wie ein Anwendungsbeispiel, das die Vorzüge der indizierten Adressierung offenkundig macht. Wer bereits jetzt mit den ix- und iy-Registern anfängt zu experimentieren, sei jedoch gewarnt: Insbesondere Manipulationen des ix-Registerinhalts können den ZX 81 zum Bocken bringen, freilich nur dann, wenn das entsprechende Maschinenprogramm im SLOW-Modus abgearbeitet wird. Im FAST-Modus bleibt der ZX 81 friedlich. Was ihn im SLOW-Modus so durcheinanderbringt, klärt ebenfalls der nächste Teil.

Klaus Herklotz

*(Wird fortgesetzt)*



**16-Bit-Befehlssatz der Indexregister:** Mit den Indexregistern sind alle 16-Bit-Operationen erlaubt, die auch beim hl-Registerpaar zugelassen sind

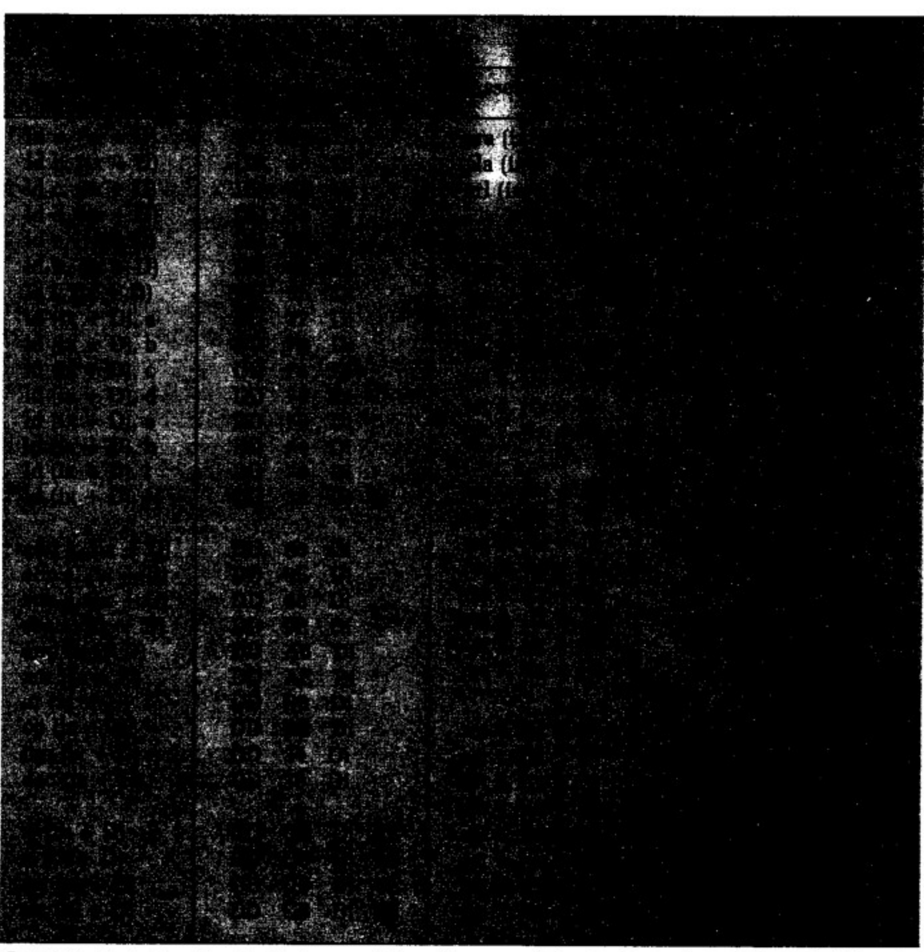
Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 24: Das ROM wird angezapft

**Im Betriebssystem des ZX 81 stecken viele Routinen, die wir für unsere Zwecke einspannen können. Zuvor schließen wir jedoch das Kapitel Indexregister ab.**

Alle indizierbaren 8-Bit-Befehle für die Indexregister zeigt *Bild 1*. Eigenartig sind dabei die Opcodes der Befehle *ld (ix + D), N* bzw. *ld (iy + D), N:* Scheinbar entgegen jeglicher Logik erscheint im Opcode das Indexbyte zuerst. Generell gilt jedoch, daß das Indexbyte immer an dritter Stelle stehen muß.



① **8-Bit-Befehle für die Indexregister:** Das angehängte Datenbyte Di ermöglicht die indizierte Adressierung. Mit FD anstelle von DD im Opcode gelten alle aufgelisteten Befehle auch für das iy-Register

### KLAR wie Klartext

Eine Möglichkeit die Nützlichkeit der Indexregister zu erproben bietet wieder der Bildspeicher: Mit Hilfe des Registers iy soll das Wort „KLAR" ins obere, linke Eck des Bildschirms geschrieben werden. Einen Lösungsvorschlag zeigt *Bild 2*.

Zuerst wirft *push iy* eine Kopie des iy-Registers auf den Stapel. Danach wird es mit dem Inhalt der Systemvariablen D-FILE geladen. Die Buchstabencodes selbst werden schließlich mit Hilfe der indizierten Adressierung in die entsprechenden Bildspeicherzellen geladen. Den Abschluß des Programms bildet der Befehl *pop iy* – der die alten Zustände im Register iy wiederherstellt – zusammen mit der Rücksprunganweisung *ret*.

### Gefahren lauern überall

Die Sache mit den Indexregistern hat freilich einen fürchterlichen Haken! Unsere CPU erhält nämlich im SLOW-Modus 50mal in der Sekunde vom SCL-Chip (Sinclair Computer Logic) die Aufforderung, den Programmablauf zu unterbrechen und sich dem Bildschirmaufbau zu widmen.

Für die Fernsehbildausgabe braucht der Prozessor jedoch unbedingt die Indexregisterpaare samt ihrem Inhalt! Das ist auch der Grund dafür, warum wir das benötigte Indexregister vor seiner Manipulation auf den Stapel gerettet haben (Bild 2).

Sollte der SCL-Baustein mit seiner Aufforderung jedoch mitten in unser Maschinenprogramm platzen, dann ist alles zu spät: Aus Verzweiflung über den falschen Wert im Indexregister „hängt" sich der Computer auf! Fürs iy-Register (im Programm nach Bild 2) kann dies durch

```
200 LET L=USR 16514
210 GOTO 200
GOTO 200
```

provoziert werden. Im FAST-Modus erledigt der Computer alles zur vollsten Zufriedenheit. Im SLOW-Modus wird er jedoch früher oder später innerhalb der Maschinenroutine von einer Aufforderung zur Fernsehbildausgabe überrascht. Nur ein Ziehen des Netzsteckers bringt dann Erlösung für alle Beteiligten.

Das iy-Register ist sogar noch etwas weniger störanfällig, als das ix-Register. Es enthält normalerweise die Hexzahl 4000 – die Anfangsadresse der Systemvariablen im RAM.

## Mit Restart-Befehlen landen wir im ROM

Wenn wir unseren ZX 81 einschalten, werden alle Register mit 00h geladen – auch der Programmzähler pc. Die CPU beginnt dann die Maschineninstruktionen des Betriebssystems ab Adresse 0000h im ROM auszuführen. Ähnliches geschieht, wenn wir ein Basic-Programm starten: Jede Basic-Anweisung ruft dann eine entsprechende Maschinenroutine im ROM auf, die ihrerseits noch Unterroutinen aufrufen kann. Die Verflechtungen sind selbst bei einem kommentierten „ROM-Listing" kaum mehr zu überschauen. So ein Buch (z. B. „Das ZX-81-ROM", Max-Huber-Verlag, Ismaning) ist aber dennoch nützlich, denn die Unterroutinen ersparen uns viel Denk- und Programmierarbeit.

Die fertigen Maschinenprogramme im ROM von Adresse 0 bis 2000h sind fehlerfrei und funktionieren immer. Um die ersten Unterroutinen im ROM aufzurufen, brauchen wir nicht einmal *call*-Befehle. Spezielle Restart-Befehle verkürzen dann nämlich den Programmieraufwand von drei Byte auf Eines: *rst 00h* mit dem Operationscode C7h entspricht dabei dem Befehl *call 0000h*. Alle weiteren Restarts lauten:

*rst 08h* (Opcode CFh) für *call 0008h*
*rst 10h* (Opcode D7h) für *call 0010h*
*rst 18h* (Opcode DFh) für *call 0018h*
*rst 20h* (Opcode E7h) für *call 0020h*
*rst 28h* (Opcode EFh) für *call 0028h*
*rst 30h* (Opcode F7h) für *call 0030h*
*rst 38h* (Opcode FFh) für *call 0038h*

Sämtliche Unterprogrammaufrufe mit *call* sind absolut adressiert. Bei den Restarts liegt dagegen eine Zero-Page-(engl.: Seite-Null-)Adressierung vor. Das höherwertige Byte der Zieladresse ist dabei stets automatisch 00h.

Ob die Routinen im ROM die gesetzten Erwartungen auch erfüllen, wollen wir zuerst an einem einfachen Problem ausprobieren. Das Basic-Programm

```
10 FOR N=1 TO 704
20 PRINT CHR$ 128;
30 NEXT N
```

bemalt den Bildschirm mit lauter schwarzen Quadraten. Das gleiche soll nun ein Maschinenprogramm ausführen: Einmal mit den Routinen im ROM und einmal ohne sie.

*Bild 3* zeigt oben die Lösung ohne ROM-Hilfe. Das de-Registerpaar erhält die Anzahl der Bildspeicherzellen (02C0h = 704), während das hl-Registerpaar die Anfangsadresse des Bildspeichers übernimmt. Bei Adresse 4088h beginnt dann die Schleife mit einem Inkrementieren des Registers hl.

Die nächsten beiden Befehle testen, ob die angesteuerte Bildspeicherzelle einen NEWLINE-Code enthält und deshalb übersprungen werden muß (408Ch). Jede andere Zelle erhält dagegen den Code des schwarzen Grafik-Zeichens zugewiesen (408Eh).

Ein 16-Bit-Dekrement-Befehl verringert schließlich den Schleifenzähler de um 1, ohne die Flags zu beeinflussen.

| ADRESSE | BYTES | | | | Z-80-ASSEMBLER |
|---|---|---|---|---|---|
| 4082 | FD | E5 | | | push iy |
| 4084 | FD | 2A | 0C | 40 | ld iy,(400C) |
| 4088 | FD | 36 | 01 | 30 | ld(iy+01),30 |
| 408C | FD | 36 | 02 | 31 | ld(iy+02),31 |
| 4090 | FD | 36 | 03 | 26 | ld(iy+03),26 |
| 4094 | FD | 36 | 04 | 37 | ld(iy+04),37 |
| 4098 | FD | E1 | | | pop iy |
| 409A | C9 | | | | ret |

② **Schreibroutine:** Nicht alle Bildspeicherzellen sind damit erreichbar

| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | 11 | C0 | 02 | | | ld de,02C0 |
| 4085 | 2A | 0C | 40 | | | ld hl,(400C) |
| 4088 | 23 | | | | LOOP | inc hl |
| 4089 | 3E | 76 | | | | ld a,76 |
| 408B | BE | | | | | cp (hl) |
| 408C | 28 | FA | | | | jr z,LOOP |
| 408E | 36 | 80 | | | | ld (hl),80 |
| 4090 | 1B | | | | | dec de |
| 4091 | 7A | | | | | ld a,d |
| 4092 | B3 | | | | | or e |
| 4093 | 20 | F3 | | | | jr nz,LOOP |
| 4095 | C9 | | | | | ret |

| ADRESSE | BYTES | | | | LABEL | Z-80-ASSEMBLER |
|---|---|---|---|---|---|---|
| 4082 | 11 | C0 | 02 | | | ld de,02C0 |
| 4085 | 3E | 80 | | | LOOP | ld a,80 |
| 4087 | D7 | | | | | rst 10 |
| 4088 | 1B | | | | | dec de |
| 4089 | 7A | | | | | ld a,d |
| 408A | B3 | | | | | or e |
| 408B | 20 | F8 | | | | jr nz,LOOP |
| 408D | C9 | | | | | ret |

③ **Schwarzmaler:** Der Bildschirm wird schwarz ausgemalt. Ein Restart-Befehl (unten) spannt für die gleiche Aufgabe das ROM mit ein

Daher muß eine logische ODER-Verknüpfung festellen, ob der Zählerstand schon 0 ist. Wenn nicht, wird die Schleife wiederholt, ansonsten geht's zurück ins Basic.

Nach dem POKE-Vorgang bringt das Listing gleich eine Überraschung mit sich: Die REM-Zeile mit dem Maschinencode ist wie abgeschnitten (das Programm ist jedoch lauffähig). Schuld daran trägt das Datenbyte 76h (NEWLINE-Code). Der Prozessor legt es als Ende von Zeile und Listing aus. Erst LIST 20 holt den Rest des Listings auf den Bildschirm. Durch die Bytefolge „3E3B07" statt „3E76" könnte der „Zwangslöschung" abgeholfen werden (Vorsicht: Auch das Sprungbyte muß verändert werden!).

RAND USR 16514 erledigt jetzt die Aufgabe genauso zuverlässig wie das Basic-Programm, nur um ein Vielfaches schneller. Eine Vereinfachung des Ganzen bietet das ROM-Unterprogramm PRINT bei Adresse 10h. Es gibt den Inhalt des Akkumulators aus, berücksichtigt dabei das Zeilenende und verändert die PRINT-Position im Bildspeicher. Unser Programm wird damit kürzer (*Bild 3, unten*):

Registerpaar de erhält wieder den Startwert 02C0h. Gleich danach beginnt schon die Schleife. Der Akku erhält den Code des schwarzen Grafik-Zeichens (4085h), wonach die Druckroutine mit einem Restart aufgerufen wird. Der restliche Teil ist mit dem vorherigen Programm identisch.

Wie leicht zu erkennen ist, vereinfacht die Druckroutine das Programm wesentlich. Dagegen muß man aber einen Geschwindigkeitsverlust in Kauf nehmen.

## Stop-Schild für die CLS-Routine

Ein weiteres interessantes Unterprogramm ist die CLS-(Clear-Screen-)Routine bei Adresse 0A2Ah. Dort erhält das b-Register zunächst die Anzahl der Bildschirmzeilen, die zu löschen sind. Wenn man diese erste Programmzeile übergeht, kann man die Zeilenzahl ändern. Das kurze Programm aus *Bild 4* löscht beispielsweise die unteren elf Bildschirmzeilen.

Nachdem das b-Register die Zeilenanzahl erhalten hat, ruft ein *call*-Befehl die CLS-Routine bei Adresse 0A2Ch auf. Nach dem Rücksprung ist die entsprechende Zeilenanzahl gelöscht. Dieses kurze Maschinenprogramm kann sogar in Form von ZX-81-Zeichen direkt in eine REM-Zeile eingegeben werden. Der Aufruf erfolgt wie gewöhnlich mit RAND USR 16514.

## Rauswerfen von Programmzeilen

Die Routinen im ROM können bei selbstgeschriebenen Programmen Unterstützung liefern. Weiterhin können sie helfen, das Betriebssystem unseres Computers zu verbessern. Sollen beispielsweise ganze Programmteile gelöscht werden, so mußte man bisher alle betroffenen Zeilennummern jedesmal gefolgt von NEWLINE eintippen. Viel schneller und eleganter löst ein Maschinenprogramm dieses Problem (*Bild 5*).

Dazu muß der Programmcursor auf die Zeile deuten, die als erste zu löschen ist. Die erste Programmzeile die danach erhalten bleiben soll, muß als Argument von RAND eingegeben werden. Dazu ein Beispiel: Der Programmcursor zeigt auf Zeile 300. RAND 400 gefolgt von LET L=USR 16514 löscht dann die Programmzeilen 300 bis 399. Wie aber ist das mit nur acht Programmschritten möglich?

Bei Adresse 4082h erhält das hl-Registerpaar den Inhalt der Systemvariablen E-PPC. Sie enthält die Zeilennummer der Programmzeile, in welcher der Programmcursor steht. Danach rufen wir mit *call 9D8h* das Unterprogramm ZEILENADRESSE im ROM auf. Dieses Unterprogramm ermittelt zu der vorgegebenen Zeilennummer im hl-Registerpaar die Adresse im Speicher und lädt sie ins hl-Registerpaar. Nach der Rückkehr von dieser Routine wirft *push hl* die so gewonnene Adresse auf den Stapel.

Danach erhält das hl-Registerpaar über die Systemvariable SEED die Zeilennummer zugewiesen, die auf RAND folgte (4089h). Wieder über das Unterprogramm ZEILENADRESSE erhält das hl-Registerpaar die zugehörige Adresse, und *pop de* bringt den zuvor gespeicherten Wert ins de-Registerpaar. Zu guter Letzt löscht dann das Unterprogramm SPEICHERRÜCKGABE (bei Adresse A5Dh) im Programmspeicher alle Speicherzellen zwischen hl und de, indem diese in den Bereich des freien Speicherplatzes verlegt werden. Klaus Herklotz

*(Wird fortgesetzt)*

```
4082 06 0B        ld b,0B
4084 CD 2C 0A     call CLS
4087 C9           ret
```

④ **CLS-Routine:** Der Bildspeicher läßt sich damit teilweise löschen

```
4082 2A 0A 40     ld hl,(400A)
4085 CD D8 09     call ZEILADR
4088 E5           push hl
4089 2A 32 40     ld hl,(4032)
408C CD D8 09     call ZEILADR
408F D1           pop de
4090 CD 5D 0A     call SPRUECK
4093 C9           ret
```

⑤ **Zeilenlöscher:** Zusammenhängende Zeilenblöcke werden rasch und elegant gelöscht

# Berichtigung

## Daten-Drehscheibe

*FUNKSCHAU 1984, Heft 4, Seite 75*

Bitte den Lötkolben aufheizen: Beim PIO-Port für den ZX 81 fehlt die Verbindung zwischen Pin 4 und Pin 5 (siehe auch Schaltbild in Teil 1) des Adreßdecoders 74LS138. Und eine gute Nachricht: Einige der im Beitrag erwähnten Einschränkungen im Betrieb lassen sich offenbar aufheben. Mehr darüber später.

## Einführung in Z-80-Maschinensprache:

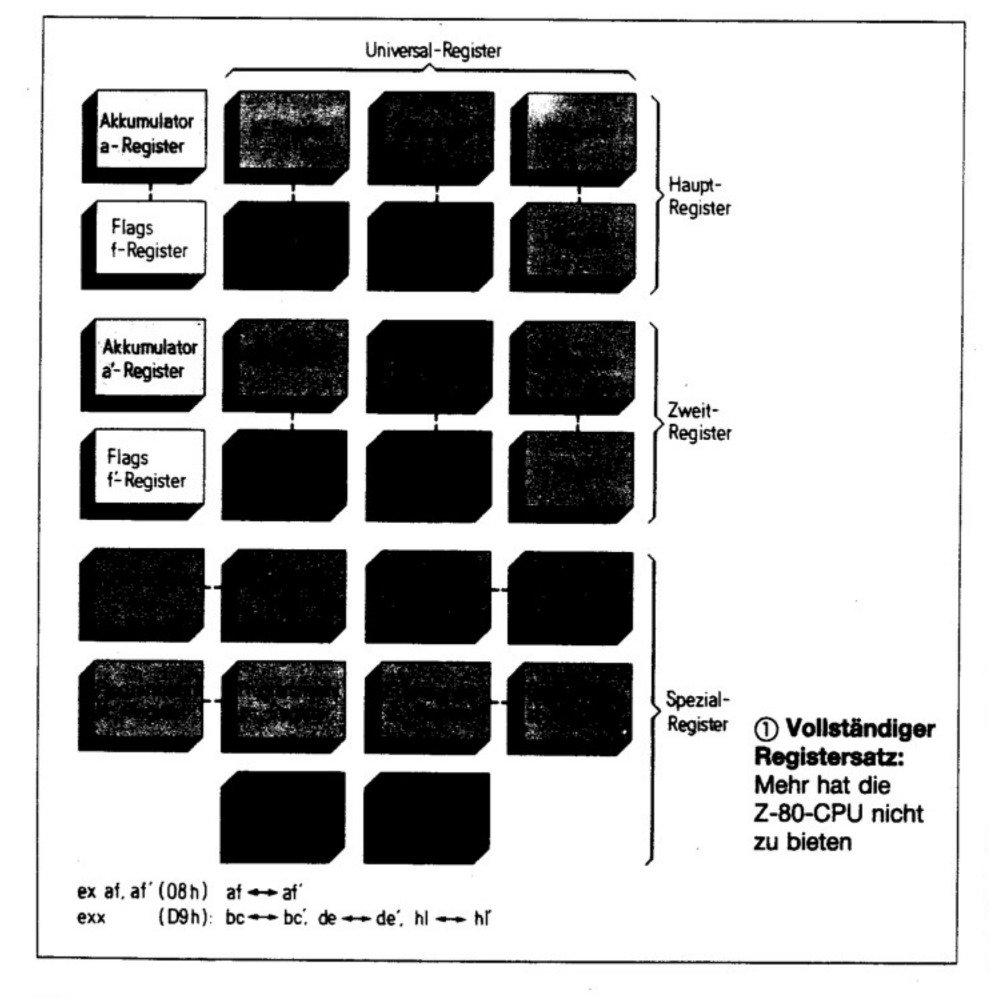
# Klartext für den ZX 81

## Teil 25: Weggabelung

**Diesmal geht es um „Interrupts". Das sind Impulse, die grundsätzlich den Programmablauf unterbrechen, wenn auf die CPU eine dringendere Aufgabe wartet.**

Neben dem bekannten Hauptregistersatz bietet die CPU noch einen Zweit- oder Schwester-Registersatz (*Bild 1*). Alle Zweit-Register sind mit einem Apostroph gekennzeichnet. Theoretisch kann man zwischen den beiden Registerblöcken frei wählen. Der Befehl exx vertauscht dann die Universalregister, während *ex af, af'* die beiden af-Registerpaare vertauscht. Es ist beim ZX 81 aber dringend davon abzuraten, die Zweit-Register zu manipulieren!

Außer dem Zweit-Registersatz sind schließlich noch die beiden Spezialregister i und r vorhanden. Das Speicherauffrischungs-Register r (Memory Refresh) dürfen wir gleich wieder vergessen, weil es zum Betrieb von dynamischen RAMs gebraucht wird, was außerhalb unseres Anwendungsbereiches liegt. Etwas mehr Bedeutung hat dagegen das i-Register als Interrupt-Vektor.




① **Vollständiger Registersatz:** Mehr hat die Z-80-CPU nicht zu bieten

### Impulse unterbrechen den Programmablauf

Ein Interrupt ist ganz einfach eine Unterbrechung des Programmablaufs. Irgendein Baustein, der mit der CPU verbunden ist, kann diese bei der Bearbeitung eines Programms unterbrechen und zu einem Unterprogramm schicken. Ein Beispiel: Alle 20 ms meldet sich der Logik-Chip bei der Zentraleinheit und fordert eine Fernsehbildausgabe. Die CPU unterbricht daraufhin die Bearbeitung des laufenden Programms, erledigt durch ein Unterprogramm (Interrupt-Behandlungsroutine) die Fernsehbildausgabe und fährt erst danach mit dem Programm fort.

Damit ein Baustein den Programmablauf unterbrechen kann, muß er mit einer der drei Leitungen $\overline{\text{INT}}$, $\overline{\text{NMI}}$ oder $\overline{\text{BUSRQ}}$ verbunden sein. Der Komplementstrich über den einzelnen Buchstaben besagt, daß im Normalzustand volle Spannung (5 V) anliegt und der Impuls durch kurzzeitige Wegnahme der Spannung erzeugt bzw. erkannt wird. Man spricht in diesem Fall von Aktiv-Low-Eingängen.

#### Finger weg vom BUSRQ!

Der „Bus Request" (BUSRQ) ist ein Interrupt-Verfahren, von dem jeder Programmierer die Finger lassen sollte. Ein Impuls auf die $\overline{\text{BUSRQ}}$-Leitung erzeugt eine solche Unterbrechung. Beim ZX 81 wird der $\overline{\text{BUSRQ}}$ nicht verwendet. Weitere Erklärungen können wir uns daher ersparen.

Interessant ist nur die Priorität: Ein Bus Request ist so dringend, daß nicht einmal das Ende des gerade bearbeiteten Befehls abgewartet wird (*Bild 2*).

#### Nicht zu bremsen: der NMI

Ein Aktiv-Low-Impuls auf der $\overline{\text{NMI}}$-Leitung löst den nichtmaskierbaren Interrupt (NMI) aus. Bei dieser Sorte von Interrupt unterbricht der Computer das laufende Programm und führt den Befehl *rst 66h* aus.

Ein NMI wird nur am Ende eines Maschinenbefehls wahrgenommen und nur dann bearbeitet, wenn kein Bus Request ansteht! Beim ZX 81 ist die NMI-Leitung an der CPU mit dem Ausgang des NMI-Generators am Logik-Chip verbunden.

Im FAST-Modus ist der NMI-Generator ausgeschaltet, im SLOW-Modus erzeugt er dagegen Aktiv-Low-Impulse, so daß der Prozessor alle 20 ms zur Maschinenroutine bei Adresse 0066h im ROM geschickt wird. Dort wird dann die Ausgabe eines Fernsehbildes vorbereitet.

### INT: der komfortable Interrupt

Die gewöhnlichen Interrupts (INT) sind vom Programmierer maskierbar, das heißt, die CPU beachtet einen Aktiv-Low-Impuls auf der INT-Leitung auf Wunsch des Programmierers nicht. Dazu muß man „Interrupt-Flipflops" (IFF) setzen. Sie stellen eine Art Flag dar: Melden die IFF logisch 1, werden ankommende Interrupts beachtet, melden sie dagegen logisch 0, so werden Interrupts unterdrückt.

Die beiden Befehle *ei* (enable Interrupts: Interrupts freigeben) und *di* (disable Interrupts: Interrupts sperren) setzen die Flipflops wunschgemäß.

Einen gewöhnlichen Interrupt akzeptiert der Prozessor nur am Ende eines Maschinenbefehls und führt ihn nur dann aus, wenn weder ein nichtmaskierbarer Interrupt noch ein Bus Request ansteht.

Gesetzt den Fall, all diese Bedingungen sind erfüllt, so gibt es drei verschiedene Arten, den gewöhnlichen Interrupt auszuwerten.

○ Interrupt-Modus 0: Wurde der Prozessor durch den Befehl *im 0* in diesen Modus geschaltet und ein Baustein legt einen Impuls auf die INT-Leitung der CPU, dann zeigt die CPU diesem Gerät durch ein IORQ-Signal, daß sie die Unterbrechung akzeptiert hat. Das Gerät muß nun den Operationscode dieses Befehls auf den Datenbus legen. Den dazugehörigen Befehl führt die CPU augenblicklich aus! Sollte der Befehl mehrere Bytes lang sein, liest die CPU alle weiteren Bytes nacheinander.

○ Interrupt-Modus 1: Dieser Modus läßt sich durch *im 1* wählen. Im Falle eines Interrupts unterbricht die CPU das laufende Programm und führt den Befehl *rst 38h* aus. Im ROM des ZX 81 befindet sich dort ein weiteres Unterprogramm zum Bildschirmaufbau.

○ Interrupt-Modus 2: Bei diesem Modus tritt das i-Register in Aktion. Kommt es jetzt zu einem Interrupt, ermittelt der Prozessor eine Speicherzelle MM. In die Speicherzellen MM und MM+1 muß der Programmierer vorher die Anfangsadresse eines Unterprogramms geladen haben (indirekte Adressierung; darum MSB zuletzt!). Die CPU führt umgehend einen entsprechenden *call*-Befehl aus. Die 16-Bit-Adresse MM (auch Interrupt-Vektor genannt) setzt sich aus den 8 Bit des i-Registers, 7 Bit vom Gerät (über den Datenbus) sowie einer abschließenden logischen 0 zusammen (Bild 2).

Beim ZX 81 enthält das i-Register das Byte 1Eh. Damit deutet der Vektor stets auf den Zeichengenerator, der bei Adresse 1E00h im ROM beginnt.

Behandlungsroutinen des maskierbaren Interrupts sollte der Befehl *reti* (Return from Interrupt), Routinen des NMI sollte der Befehl *retn* (Return from Non Maskable Interrupt) abschließen. Neben diesen beiden Befehlen gibt es im Befehlssatz noch den Befehl *halt:* Er veranlaßt den Prozessor, so lange *nop*-Befehle auszuführen, bis ein Interrupt anliegt.

Alle Interrupt-Leitungen sind unter anderem an der Rückseite des ZX 81 herausgeführt. Über diese Schnittstelle kann der Computer Verbindung zu anderen Geräten aufnehmen. Will beispielsweise ein externes Gerät vom Computer „behandelt" werden, kann dieses Gerät durch Interrupt-Impulse auf sich aufmerksam machen. Wenn das Gerät außerdem den hinteren Adressenteil des Interrupt-Vektors liefert, dann kann es im Interrupt-Modus 2 eine individuelle Behandlungsroutine erzwingen.

Beim ZX 81 werden die Interrupts für systemspezifische Zwecke benötigt. Sinnvolle Anwendungen ohne einen Hardwarezusatz (z. B. PIO: siehe FUNKSCHAU 3/84) gibt es nicht.

Klaus Herklotz

*(Wird fortgesetzt)*

ei FB
di F3
im 0 ED 46
im 1 ED 56
im 2 ED 5E

halt 76
reti ED 4D
retn ED 45

BUS RQ → direkter Speicherzugriff

NMI → rst 66h

INT → im0 Befehl vom Datenbus; im1 rst 38 h; im2 push pc

ld pcL/ (0) ld pcH/ (1) i-Register Gerät

② **Interrupt-Techniken:** Die Übersicht zeigt, wie die CPU reagiert, wenn ein Interrupt in den Programmteil mit den Bytes ...3E 29 21 40... hineinplatzt

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 26: Die CPU geht fremd

**Unsere Serie neigt sich dem Ende zu. Da darf die CPU schon einmal fremdgehen – und den Datenverkehr mit externen Geräten aufnehmen.**

Damit ein interner Datenfluß zwischen den einzelnen Speicherzellen und den Registern der CPU überhaupt möglich ist, muß praktisch jede einzelne Speicherzelle und jedes Register mit dem Datenbus verbunden sein. Für die gezielte Adressierung einer Speicherzelle sind dann die sechzehn Adreßleitungen zuständig (*Bild 1*).

### Der Logik-Chip spielt den Vermittler

Die CPU legt beispielsweise beim Befehl *ld a,(4081h)* die nachstehenden 16 Spannungszustände (Bits)

0100 0000 1000 0001 (4081h)

auf die Leitungen des Adreßbusses. Weiterhin meldet sie dem Logik-Chip durch das Signal MREQ (Memory-Request: Speicher-Aufforderung), daß am Adreßbus die Adresse bereitliegt. Der Logik-Chip meldet die Speicher-Aufforderung über die Leitungen RAM-CS bzw. ROM-CS (CS = Chip-Select: Chip-Anwahl) gleich an den entsprechenden Speicherbaustein weiter (*Bild 1*).

Der Speicherbaustein legt nun den Inhalt der adressierten Speicherzelle auf den Datenbus, den die CPU mittlerweile mit dem Akku verbunden hat: Der Inhalt der Speicherzelle 4081h steht damit im Akku.

Ein weiteres Beispiel: Beim Befehl *ld (hl),b* schaltet die CPU das hl-Registerpaar auf den Adreß- und das b-Register auf den Datenbus. Und wieder zeigt die MREQ-Leitung dem Speicher direkt an, daß eine für ihn gültige Adresse am Adreßbus liegt.

Wie aber erfährt der Speicher, ob die CPU Daten von ihm haben will (erstes Beispiel) oder ob er Daten speichern soll (zweites Beispiel)? Damit es darüber keine Mißverständnisse gibt, sind aus der CPU die beiden Leitungen WR (Write: Schreiben) und RD (Read: Lesen) herausgeführt. Mit einem WR-Signal meldet die CPU, daß der Datenbus ein Byte bereithält; mit einem RD-Signal fordert sie ein Byte über den Datenbus an. Normalerweise genügt, wie beim ZX 81, einem RAM-Speicher-IC bereits eines der beiden Signale, z. B. WR. Abhängig von dessen logischen Pegel, wird dann die Schreib- (WR = 0) oder Lese-Operation (WR = 1) ausgeführt.

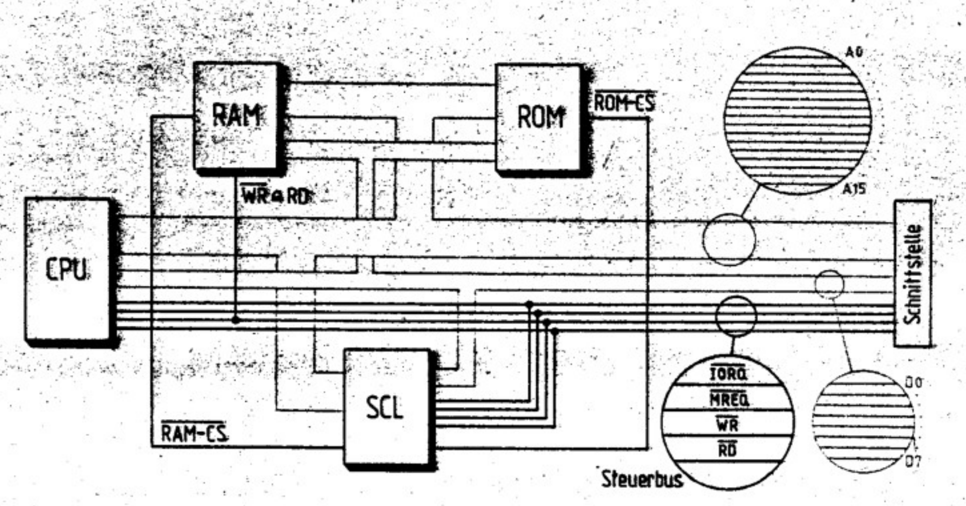
Problematisch wird's jedoch, wenn der Computer kein Speicher-IC, sondern irgendeinen Baustein außerhalb des Systems mit Daten versorgen soll. Als unmißverständliche Anzeige dafür dient die Leitung IORQ (Input/Output-Request: Ein-/Ausgabe-Aufforderung): Im Gegensatz zur MREQ-Leitung zeigt sie es an, wenn der Datenverkehr einen externen Baustein betrifft.

### Zwei Befehle öffnen das Tor zur Umwelt

Nehmen wir einfach an, der ZX 81 soll ein Gerät regeln, z. B. einen Heizkörper. Was liegt dann näher, als die Heizungsregelung über den Adreßbus zu adressieren (anzusteuern) und dabei über den Datenbus mit Daten (z. B. über die Temperatur) zu versorgen?

Die CPU kann auf diese Weise für gewöhnlich 256 Geräte ansprechen, wenn man zur Adressierung lediglich die Adreßleitungen A0 bis A7 (auch Kanal genannt) verwendet. Dann ist z. B. ein Gerät mit der Adresse *F9h* gerade so an den Adreßbus anzuschließen, daß es nur durch ein IORQ-Signal während des Zustands *F9h* auf dem Adreßbus angesteuert wird. Meist erledigt letzteres ein Adreßdecoder-Baustein, der feststellt, wann der Adreßbus die erwünschte Adresse bereithält und diese Information in Form eines CS-Signals an das Gerät weiterleitet.

Weiterhin muß das Gerät mit dem Datenbus verbunden sein, damit es im angesteuerten Zustand Daten aufnehmen kann. Speziell dafür stellt der Z-80-Befehlssatz die Ein- und Ausgabebefehle zur Verfügung (*Bild 2*): Der Ausgabebe-




① **Computersystem:** Der ZX 81 besteht im Prinzip aus lediglich vier Bausteinen. Die Skizze zeigt nur die fürs Verständnis notwendigen Leitungen

fehl *out (n),a* schaltet die 8-Bit-Adresse n auf die untere Hälfte des Adreßbusses (A0 bis A7), erzeugt ein $\overline{IORQ}$- sowie ein $\overline{WR}$-Signal und er legt den Akkuinhalt auf den Datenbus, gleichzeitig aber auch auf die obere Hälfte des Adreßbusses (A7 bis A15). Dazu gleich ein Beispiel:

Der Befehl *ld a,4F* gefolgt vom Befehl *out (F9),a* erzeugt am Datenbus die Pegelverteilung 0100 1111 (4Fh) und am Adreßbus die Pegelverteilung
0100 1111 1111 1001 (4FF9h).

Umgekehrt kann die CPU auch Daten von einem Gerät beziehen: Dazu legt *in a,(n)* die 8-Bit-Adresse n auf die untere Hälfte vom Adreßbus und fordert das Gerät durch ein $\overline{IORQ}$- sowie ein $\overline{RD}$-Signal auf, Daten auf den Datenbus zu legen.

| Befehl | | Flags | | | | | | Erklärung |
|---|---|---|---|---|---|---|---|---|
| Assemblercode | Opcode | S | Z | H | P/V | N | C | |
| [illegible] | D3 ... | — | — | — | — | — | — | n → $A_0 \ldots A_7$<br>a → $A_8 \ldots A_{15}$<br>a → $D_0 \ldots D_7$ |
| in a, (n) | DB ... | — | — | — | — | — | — | n → $A_0 \ldots A_7$<br>a → $A_8 \ldots A_{15}$<br>$D_0 \ldots D_7$ → a |
| [illegible] | ED 79<br>ED 41<br>ED 49<br>ED 51<br>ED 59<br>ED 61<br>ED 69 | — | — | — | — | — | — | c → $A_0 \ldots A_7$<br>b → $A_8 \ldots A_{15}$<br>r → $D_0 \ldots D_7$<br>mit r = a,b,c,d,e,h,l |
| in a, (c)<br>in b, (c)<br>in c, (c)<br>in d, (c)<br>in e, (c)<br>in h, (c)<br>in l, (c) | ED 78<br>ED 40<br>ED 48<br>ED 50<br>ED 58<br>ED 60<br>ED 68 | ! | ! | ! | ! | 0 | — | c → $A_0 \ldots A_7$<br>b → $A_8 \ldots A_{15}$<br>$D_0 \ldots D_7$ → r<br>mit r = a,b,c,d,e,h,l |
| [illegible] | ED A3 | ? | ! | ? | ? | 1 | — | out (c),(hl)<br>inc hl ; dec b |
| otir | ED B3 | ? | 1 | ? | ? | 1 | — | outi<br>Wdh. bis b=0 |
| [illegible] | ED AB | ? | ! | ? | ? | 1 | — | out (c),(hl)<br>dec hl ; dec b |
| otdr | ED BB | ? | 1 | ? | ? | 1 | — | outd<br>Wdh. bis b=0 |
| [illegible] | ED A2 | ? | ! | ? | ? | 1 | — | in (hl),(c)<br>inc hl ; dec b |
| inir | ED B2 | ? | 1 | ? | ? | 1 | — | ini<br>Wdh. bis b=0 |
| [illegible] | ED AA | ? | ! | ? | ? | 1 | — | in (hl),(c)<br>dec hl ; dec b |
| indr | ED BA | ? | 1 | ? | ? | 1 | — | ind<br>Wdh. bis b=0 |

Zeichenerklärung:
! Flag wird entsprechend beeinflußt
? Flag wird willkürlich beeinflußt
1 Flag wird gesetzt
0 Flag wird zurückgesetzt
— Flag wird nicht beeinflußt

② **Ein-/Ausgabebefehle:** Mit dieser Befehlsgruppe ist der Z-80-Befehlssatz vollständig

Um jetzt einen Ausgabebefehl auszuprobieren, muß man nicht unbedingt ein Gerät (an die Schnittstelle) anschließen. Man kann auch auf die bereits im ZX 81 vorhandenen „Geräte" zurückgreifen: Bekanntlich befindet sich im Logik-Chip der $\overline{NMI}$-Generator, der im SLOW-Modus regelmäßig $\overline{NMI}$-Impulse erzeugt und den Computer damit zur Fernsehbildausgabe zwingt. Soll der Computer dagegen im FAST-Modus laufen, muß man bloß diesen Generator abschalten! Der nötige Aus-Schalter hat die Adresse *FDh* und läßt sich mit einem Ausgabebefehl aktivieren.

Das kurze Maschinenprogramm aus *Bild 3* schaltet den Rechner vom SLOW- in den FAST-Modus. Zuerst wartet *halt* die nächste Fernsehbildausgabe ab, wartet also auf einen Interrupt (Adresse 4082h im Listing). Gleich danach schaltet ein Ausgabebefehl den $\overline{NMI}$-Generator ab (4083h). Es ist hier völlig gleichgültig, welcher Wert im Akku steht, denn der Ausgabebefehl wird hier nicht zur Datenübertragung, sondern für einen einfachen Schaltvorgang verwendet.

Damit das System anschließend nicht zusammenbricht, muß der Computer über den Schaltvorgang informiert werden: Zwei Einzelbit-Befehle (4085h; 4089h) löschen dazu Bit 6 und 7 der Systemvariablen CDFLAG, die bei Adresse 16443 (403Bh) zu finden ist. Die indizierte Adressierung ist hier möglich, weil das iy-Register stets den Wert 4000h aufweist.

Wie Bild 2 noch zeigt, sind die Ein- und Ausgabebefehle auch mit dem bc-Registerpaar adressierbar. Der Befehl *out (c),r* legt den Inhalt des bc-Registerpaares auf den Adreßbus und den Inhalt des Registers r auf den Datenbus. Dementsprechend arbeitet auch der Eingabebefehl *in r,(c)*. Bemerkenswert ist noch, daß die Flags nur von register-adressierten Eingabebefehlen beeinflußt werden! Weiterhin ist die Ein- und Ausgabe von Daten per Blocktransfer möglich. Einzelheiten dazu zeigt Bild 2.

Wer mit seinem Computer Steuerungsaufgaben durchführen will, kann von PIO-Bausteinen (Parallel Input Output Interface Controller) Gebrauch machen (siehe FUNKSCHAU 3/1984). Diese lassen sich mit den Ein-/Ausgabebefehlen steuern.

Klaus Herklotz

*(Schluß folgt)*

```
ADRESSE  BYTES            Z-80-ASSEMBLER
4082     76               halt
4083     D3 FD            out (FD),a
4085     FD CB 3B B6      res 6,(iy+3B)
4089     FD CB 3B BE      res 7,(iy+3B)
408D     C9               ret
```

③ **Geschwindigkeitswechsel:** Das Programm darf nur im SLOW-Modus mit RAND USR 16514 aufgerufen werden!

## Einführung in Z-80-Maschinensprache:

# Klartext für den ZX 81

## Teil 27: Schlußakkorde

**Die letzten noch unbekannten Z-80-Befehle haben wir bereits im vorangegangenen Teil kennengelernt. Wir können es uns daher leisten, die Serie mit diesem Schlußteil „gemütlich" ausklingen zu lassen: Reden wir noch einmal im Klartext miteinander.**

Ein Kompliment an diejenigen, die bis zu diesem Teil durchgehalten haben! Sie beherrschen die Z-80-Maschinensprache freilich jetzt schon so gut, daß in Zeitschriften und Büchern veröffentlichte Assembler-Listings kaum noch Kummerfalten hervorrufen können. Und der eigenen Experimentierlust zum Lösen der unmöglichsten Probleme steht nunmehr Tür und Tor offen. Eine Schwierigkeit wollen wir aber noch gemeinsam aus dem Weg räumen: Die Dateneingabe über die Tastatur, während des Ablaufs eines Maschinenprogramms.

```
4082 CD A3 40    START   call TASTE
4085 28 FB               jr z,START
4087 44                  ld b,h
4088 4D                  ld c,l
4089 CD BD 07            call DECOD
408C 7E                  ld a,(hl)
408D FE 00               cp 00h
408F C8                  ret z
4090 FE 1C               cp 1Ch
4092 FA 82 40            jp m,START
4095 FE 2C               cp 2Ch
4097 F2 82 40            jp p,START
409A D7                  rst 10h
409B CD A3 40    WARTE   call TASTE
409E 20 FB               jr nz,WARTE
40A0 C3 82 40            jp START
40A3 CD BB 02    TASTE   call ABFRAGE
40A6 3E FF               ld a,FFh
40A8 BD                  cp l
40A9 C9                  ret
```

**① Dateneingabe:** Dieses Programm zeigt, wie innerhalb eines Maschinenprogramms eine Tastaturabfrage möglich ist

Bisher mußte dazu die Systemvariable LAST-K (siehe Teil 2 und Teil 11) ihren Inhalt zur Verfügung stellen. Dies führt aber oft zu Problemen, weil sich nur umständlich feststellen läßt, welche Taste gedrückt wurde. Eine bessere Lösung ermöglichen zwei ROM-Routinen bei der Adresse 699 = 2BBh und bei der Adresse 1981 = 7BDh.

Bei Adresse 2BBh im ROM befindet sich eine Maschinenroutine zur Tastaturabfrage. Sie legt den gegenwärtigen „Zustand" der Tastatur, in Form eines verschlüsselten 16-Bit-Wertes, im hl-Registerpaar ab. Sollte z. B. keine Taste gedrückt sein, erhält das hl-Registerpaar den Wert FFFFh.

Bei Adresse 7BDh befindet sich dann die dazugehörige Decodier-Routine. Diese weist dem hl-Registerpaar die Adresse im ROM zu, unter welcher das Byte des eingegebenen Zeichens – das jetzt im bc-Registerpaar verschlüsselt ist – zu finden ist. Das klingt ziemlich kompliziert, aber das folgende Anwendungsbeispiel bringt Klarheit: Der ZX 81 soll auf einen Tastendruck warten, und falls eine Hex-Ziffer (0 bis F) eingegeben wurde, diese am Bildschirm schreiben.

Das Maschinenprogramm (*Bild 1*) beginnt mit dem Aufruf des Unterprogramms TASTE bei Adresse 40A3h. Dieses Unterprogramm ruft seinerseits die ROM-Routine zur Tastaturabfrage auf. Nach der Rückkehr aus dem ROM ist dann der Tastaturzustand im hl-Registerpaar verschlüsselt. Nachfolgend stellt ein Compare-Befehl (40A8h im Listing) fest, ob eine Taste gedrückt wurde und löscht das Zero-Flag falls das der Fall war.

Bleibt das Zero-Flag gesetzt, d. h., wird keine Taste gedrückt, erfolgt sofort ein Sprung (4085h) zum Anfang des Programms. Dieser Prozedur wiederholt sich solange, bis man schließlich eine Taste drückt: Zwei Ladebefehle kopieren dann das hl- ins bc-Registerpaar und ein unbedingter *call*-Befehl ruft die Decodier-Routine auf (4089h). Ein einfacher Ladebefehl (408Ch) bringt danach den Code der gedrückten Taste in den Akku. Falls dies der BREAK-Code war, geht's zurück ins Basic (408Dh bis 408Fh).

Die nächsten Programmzeilen veranschaulichen eine interessante Programmier-Technik. Es muß ja noch festgestellt werden, ob eine – oder keine – Hex-Ziffer eingegeben wurde. In Basic würde das entsprechende Verfahren so aussehen (28 ≙ 1Ch = Code für 0; 43 ≙ 2Bh = Code für F):

```
1 LET A=CODE INKEY$
2 IF A<28 THEN RUN
3 IF A>43 THEN RUN
4 PRINT CHR$ A
5 RUN
```

In der Z-80-Maschinensprache fehlen aber die „Größer-Kleiner"-Relationen. Alle Auskünfte darüber müssen über die Flags bezogen werden: *cp 1Ch* subtrahiert dazu 28 vom Akkuinhalt und *jp m,START* sorgt für den Sprung zum Programmanfang, sofern jetzt das Vorzeichen-Flag (siehe Teil 19) gesetzt ist, weil der Akkuinhalt kleiner als 1Ch war. Selbiges besorgt *cp 2Ch* gefolgt von *jp p,START* für den Fall, daß der Akkuinhalt größer als 2Bh war.

Führen Sie sich diesen Sachverhalt unbedingt in Ruhe zu Gemüte, denn Relationen können nur über die Flags programmiert werden.

Der Rest des Programmes gibt das Zeichen am Bildschirm aus und wartet darauf, daß die gedrückte Taste wieder losgelassen wird, um ein unbeabsichtigtes „Auto-Repeat" (Wiederholfunktion) zu vermeiden. Werfen wir jetzt noch einen kurzen Blick auf die drei Eingabemöglichkeiten für Maschinencode-Programme.

## Für und Wider der Eingabemethoden

Ein Monitor erlaubt im allgemeinen die Betrachtung eines ganzen Speicherteils und hilft bei der Programm-Eingabe. Wer sich schon etwas mehr zutraut, kann auf der Basis des Maschinenprogramms aus Teil 18, einen einfachen Monitor selbst programmieren. Ansonsten kann der Monitor aus FUNKSCHAU 6/1984 nur empfohlen werden.

Am einfachsten ist und bleibt dagegen das Verfahren, ein Programm mit POKE im Speicher unterzubringen. Anhand eines Programms, das den Bildschirminhalt invertiert, lassen sich die drei Varianten dieses Verfahrens gut aufzeigen.

Im einfachsten Fall erhält ein REM-Kommentar am Anfang des Programmes die Bytes des Maschinenprogrammes zugewiesen. Der Aufruf erfolgt gewöhnlich mit RAND USR A.

Eine weitere Möglichkeit bietet der Variablenbereich (*Bild 2*). Eine DIM-Anweisung schafft *am Anfang* des Variablenbereichs ab Adresse VARS+6 Platz für den Maschinencode, der dann analog zur ersten Methode das Programm aufnimmt. Der POKE-Vorgang ist stets mit RUN einzuleiten, damit der bisherige Variablenbereich gelöscht wird. Da sich das Programm im Speicher verschieben kann, sollte es mit RAND USR (PEEK 16400+256*PEEK 16401+6) aufgerufen werden.

Als letztes kommt der Bereich über RAMTOP in Betracht. Ein Vorbereitungsprogramm (siehe Teil 20 und *Bild 3*) verringert dazu den Wert von RAMTOP um die Länge des Maschinenprogramms (Bild 3 oben). Erst danach schreibt ein Eingabeprogramm die Bytes des Strings mit dem Maschinencode ein. Die Bytes sind jetzt zwar vor NEW geschützt, lassen sich aber mit SAVE nicht auf Kassette aufzeichnen. Ein Aufruf kann hier einfach mit RAND USR A erfolgen.

Wer die gezeigten Eingabeprogramme benutzt, muß immer die entsprechenden hexadezimalen Operationscodes suchen. Damit sich der Suchaufwand in Grenzen hält, sind noch einmal alle Z-80-Befehle und ihre Auswirkungen auf die Flags in *Bild 4* aufgelistet.

Zum Umgang mit den Tabellen: Es sind praktisch alle Z-80-Befehle enthalten. Alle fett hervorgehobenen Op-Codes gelten auch für die Indexregister anstelle des hl-Registerpaars als Operanden. Dem Op-Code ist dann lediglich DDh bzw. FDh voranzustellen; ein Indexbyte muß stets an dritter Stelle erscheinen!

Die Reihenfolge der Befehle in der Flag-Tabelle richtet sich – soweit möglich – nach dem Erscheinen im Befehlssatz (Ladebefehle zuerst, Einzelbitbefehle zuletzt).

Zum Abschluß noch ein paar Worte zu den Maschinenprogrammaufrufen. Einsteiger glauben naheliegenderweise, eine Maschinenroutine bei der Adresse A müssen mit GOSUB USR A aufgerufen werden. Dem ist aber nicht so! Der Ausdruck USR A ist genauso wie alle anderen Funktionen (z. B. SQR A, LN A, SIN A,...) zu verwenden. Er ruft ein Maschinenprogramm bei Adresse A auf und liefert als Ergebnis den Inhalt des bc-Registerpaars. RAND USR A weist z. B. nach dem Rücksprung der Systemvariablen SEED den Wert im bc-Registerpaar zu. Ähnlich weist LET Q=USR A der Variablen Q diesen Wert zu.

Sehr praktisch ist auch ein Aufruf mit IF: Soll z. B. vor der Programmzeile 100 PRINT „FS" noch eine Maschinenroutine aufgerufen werden, so ersetzt man diese Zeile einfach durch
100 IF USR A THEN PRINT "FS".

So, damit wären wir am Ende der Serie angekommen. Jetzt legt Ihnen der ZX 81 gewissermaßen seine Fähigkeiten in Maschinensprache zu Füßen. Und was Sie damit anfangen können ist schon sehr viel, denn der Z 80 gehört zu den leistungsfähigsten und schnellsten 8-Bit-CPUs die es gibt. Ein weiterer Vorteil: Die Z-80-CPU ist weit verbreitet. Ihr Know how ist also nicht allein auf den ZX 81 beschränkt.

Sie stehen jetzt auf eigenen Füßen. Nutzen Sie Ihre Kenntnisse, um neue, nützliche Anwendungsbeispiele für den Computer auszukundschaften und zu verwirklichen – reden Sie Klartext mit dem ZX 81! Wir, der Autor und die Redaktion, wünschen Ihnen dabei prikkelnde Erfolgserlebnisse. Klaus Herklotz

```
10 DIM Z$(19)
20 LET A$="2A0C400E160620233E80867710F9230D20F3C9"
30 LET A=PEEK 16400+256*PEEK 16401+6
40 FOR N=0 TO LEN A$-2 STEP 2
50 POKE A+INT (N/2),16*(CODE A$(N+1)-28)+CODE A$(N+2)-28
60 NEXT N
```

② **Maschinencode im Variablenfeld:** Diese Methode erfordert keine langen REM-Zeilen, aber leider liegt die Startadresse nicht immer fest

```
10 LET A=PEEK 16388+256*PEEK 16389
20 LET A=A-19
30 POKE 16388,A-256*INT (A/256)
40 POKE 16389,INT (A/256)
50 NEW
```

③ **Maschinencode oberhalb von RAMTOP:** Oberhalb von RAMTOP sind Maschinenprogramme sicher vor NEW, aber sie lassen sich nicht auf Kassette speichern

```
20 LET A$="2A0C400E160620233E80867710F9230D20F3C9"
30 LET A=PEEK 16388+256*PEEK 16389
40 FOR N=0 TO LEN A$-2 STEP 2
50 POKE A+INT (N/2),16*(CODE A$(N+1)-28)+CODE A$(N+2)-28
60 NEXT N
```

### 8-Bit-Ladebefehle

| | a | b | c | d | e | h | l | (hl) | N |
|---|---|---|---|---|---|---|---|---|---|
| | 7F | 78 | 79 | 7A | 7B | 7C | 7D | 7E | 3E |
| | 47 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 06 |
| | 4F | 48 | 49 | 4A | 4B | 4C | 4D | 4E | 0E |
| | 57 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 16 |
| | 5F | 58 | 59 | 5A | 5B | 5C | 5D | 5E | 1E |
| | 67 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 26 |
| | 6F | 68 | 69 | 6A | 6B | 6C | 6D | 6E | 2E |
| | 77 | 70 | 71 | 72 | 73 | 74 | 75 | | 36 |
| | | | 0A | | | | | 02 | |
| | | | 1A | | | | | 12 | |
| | | | 3A | | | | | 32 | |
| | | | ED57 | | | | | ED47 | |
| | | | ED5F | | | | | ED4F | |

### 16-Bit-Ladebefehle

| | bc | de | hl | sp | af |
|---|---|---|---|---|---|
| | 01 | 11 | 21 | 31 | |
| | ED4B | ED5B | 2A | ED7B | |
| | ED43 | ED53 | 22 | ED73 | |
| | C5 | D5 | E5 | | F5 |
| | C1 | D1 | E1 | | F1 |
| | F9 | | | | |

### 8-Bit-Arithmetik- und Logikbefehle

| | a | b | c | d | e | h | l | (hl) | N |
|---|---|---|---|---|---|---|---|---|---|
| | 87 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | C6 |
| | 8F | 88 | 89 | 8A | 8B | 8C | 8D | 8E | CE |
| | 97 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | D6 |
| | 9F | 98 | 99 | 9A | 9B | 9C | 9D | 9E | DE |
| | A7 | A0 | A1 | A2 | A3 | A4 | A5 | A6 | E6 |
| | AF | A8 | A9 | AA | AB | AC | AD | AE | EE |
| | B7 | B0 | B1 | B2 | B3 | B4 | B5 | B6 | F6 |
| | BF | B8 | B9 | BA | BB | BC | BD | BE | FE |
| | 3C | 04 | 0C | 14 | 1C | 24 | 2C | 34 | |
| | 3D | 05 | 0D | 15 | 1D | 25 | 2D | 35 | |

### 16-Bit-Arithmetikbefehle

| | bc | de | hl | sp |
|---|---|---|---|---|
| | 09 | 19 | 29 | 39 |
| | ED4A | ED5A | ED6A | ED7A |
| | ED42 | ED52 | ED62 | ED72 |
| | 03 | 13 | 23 | 33 |
| | 0B | 1B | 2B | 3B |

**④ Befehlstabellen für die Z-80-CPU:** Die hervorgehobenen Op-Codes gelten mit vorangestelltem DD bzw. FD auch für die Indexregister

### Rotations- und Schiebebefehle

| | a | b | c | d | e | h | l | (hl) |
|---|---|---|---|---|---|---|---|---|
| | CB07 | CB00 | CB01 | CB02 | CB03 | CB04 | CB05 | CB06 |
| | CB0F | CB08 | CB09 | CB0A | CB0B | CB0C | CB0D | CB0E |
| | CB17 | CB10 | CB11 | CB12 | CB13 | CB14 | CB15 | CB16 |
| | CB1F | CB18 | CB19 | CB1A | CB1B | CB1C | CB1D | CB1E |
| | CB27 | CB20 | CB21 | CB22 | CB23 | CB24 | CB25 | CB26 |
| | CB2F | CB28 | CB29 | CB2A | CB2B | CB2C | CB2D | CB2E |
| | CB3F | CB38 | CB39 | CB3A | CB3B | CB3C | CB3D | CB3E |
| | 07 | rla | | 17 | rld | | ED6F | |
| | 0F | rra | | 1F | rrd | | ED67 | |

### Sprungbefehle

| cc = | – | z | nz | c | nc | pe | po | m | p |
|---|---|---|---|---|---|---|---|---|---|
| jp cc,NN | C3 | CA | C2 | DA | D2 | EA | E2 | FA | F2 |
| call cc,NN | CD | CC | C4 | DC | D4 | EC | E4 | FC | F4 |
| ret cc | C9 | C8 | C0 | D8 | D0 | E8 | E0 | F8 | F0 |
| jr cc,DIS | 18 | 28 | 20 | 38 | 30 | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| jp (hl) | E9 | djnz | 10 | reti | ED4D | retn | ED45 |
| rst 00 | C7 | rst 10 | D7 | rst 20 | E7 | rst 30 | F7 |
| rst 08 | CF | rst 18 | DF | rst 28 | EF | rst 38 | FF |

### Ein- und Ausgabebefehle

| s = | a | b | c | d | e | h | l |
|---|---|---|---|---|---|---|---|
| in s,(c) | ED78 | ED40 | ED48 | ED50 | ED58 | ED60 | ED68 |
| out (c),s | ED79 | ED41 | ED49 | ED51 | ED59 | ED61 | ED69 |

| | | | |
|---|---|---|---|
| in a,(N) | DB | out (N),a | D3 |
| ini | EDA2 | outi | EDA3 |
| inir | EDB2 | otir | EDB3 |
| ind | EDAA | outd | EDAB |
| indr | EDBA | otdr | EDBB |

### Einzelbitbefehle

| s = | a | b | c | d | e | h | l | (hl) |
|---|---|---|---|---|---|---|---|---|
| bit 0,s | CB47 | CB40 | CB41 | CB42 | CB43 | CB44 | CB45 | CB46 |
| bit 1,s | CB4F | CB48 | CB49 | CB4A | CB4B | CB4C | CB4D | CB4E |
| bit 2,s | CB57 | CB50 | CB51 | CB52 | CB53 | CB54 | CB55 | CB56 |
| bit 3,s | CB5F | CB58 | CB59 | CB5A | CB5B | CB5C | CB5D | CB5E |
| bit 4,s | CB67 | CB60 | CB61 | CB62 | CB63 | CB64 | CB65 | CB66 |
| bit 5,s | CB6F | CB68 | CB69 | CB6A | CB6B | CB6C | CB6D | CB6E |
| bit 6,s | CB77 | CB70 | CB71 | CB72 | CB73 | CB74 | CB75 | CB76 |
| bit 7,s | CB7F | CB78 | CB79 | CB7A | CB7B | CB7C | CB7D | CB7E |
| res 0,s | CB87 | CB80 | CB81 | CB82 | CB83 | CB84 | CB85 | CB86 |
| res 1,s | CB8F | CB88 | CB89 | CB8A | CB8B | CB8C | CB8D | CB8E |
| res 2,s | CB97 | CB90 | CB91 | CB92 | CB93 | CB94 | CB95 | CB96 |
| res 3,s | CB9F | CB98 | CB99 | CB9A | CB9B | CB9C | CB9D | CB9E |
| res 4,s | CBA7 | CBA0 | CBA1 | CBA2 | CBA3 | CBA4 | CBA5 | CBA6 |
| res 5,s | CBAF | CBA8 | CBA9 | CBAA | CBAB | CBAC | CBAD | CBAE |
| res 6,s | CBB7 | CBB0 | CBB1 | CBB2 | CBB3 | CBB4 | CBB5 | CBB6 |
| res 7,s | CBBF | CBB8 | CBB9 | CBBA | CBBB | CBBC | CBBD | CBBE |
| set 0,s | CBC7 | CBC0 | CBC1 | CBC2 | CBC3 | CBC4 | CBC5 | CBC6 |
| set 1,s | CBCF | CBC8 | CBC9 | CBCA | CBCB | CBCC | CBCD | CBCE |
| set 2,s | CBD7 | CBD0 | CBD1 | CBD2 | CBD3 | CBD4 | CBD5 | CBD6 |
| set 3,s | CBDF | CBD8 | CBD9 | CBDA | CBDB | CBDC | CBDD | CBDE |
| set 4,s | CBE7 | CBE0 | CBE1 | CBE2 | CBE3 | CBE4 | CBE5 | CBE6 |
| set 5,s | CBEF | CBE8 | CBE9 | CBEA | CBEB | CBEC | CBED | CBEE |
| set 6,s | CBF7 | CBF0 | CBF1 | CBF2 | CBF3 | CBF4 | CBF5 | CBF6 |
| set 7,s | CBFF | CBF8 | CBF9 | CBFA | CBFB | CBFC | CBFD | CBFE |

### Flagbeeinflussung der Befehle

| | Flags | | | | | |
|---|---|---|---|---|---|---|
| Flagbeeinflussung der Befehle | S | Z | H | P/V | N | C |
| ld a,i; ld a,r | ! | ! | 0 | ! | 0 | – |
| add a,s; adc a,s | ! | ! | ! | ! | 0 | ! |
| sub a,s; sbc a,s; cp s; neg | ! | ! | ! | ! | 1 | ! |
| and s | ! | ! | 1 | ! | 0 | 0 |
| or s; xor s | ! | ! | 0 | ! | 0 | 0 |
| inc s | ! | ! | ! | ! | 0 | – |
| dec s | ! | ! | ! | ! | 1 | – |
| add hl,ss; add ix,ss; add iy,ss | – | – | ? | – | 0 | ! |
| adc hl,ss | ! | ! | ? | ! | 0 | ! |
| sbc hl,ss | ! | ! | ? | ! | 1 | ! |
| rla; rlca; rra; rrca | – | – | 0 | – | 0 | ! |
| rl s; rlc s; rr s; rrc s; sla s; sra s; srl s | ! | ! | 0 | ! | 0 | ! |
| rld; rrd | ! | ! | 0 | ! | 0 | – |
| daa | ! | ! | ! | ! | ? | ! |
| cpl | – | – | 1 | – | 1 | – |
| ccf | – | – | ? | – | 0 | ! |
| scf | – | – | 0 | – | 0 | 1 |
| in s,(c) | ! | ! | 0 | ! | 0 | – |
| ini; ind; outi; outd | ? | ! | ? | ? | 1 | – |
| inir; indr; otir; otdr | ? | 1 | ? | ? | 1 | – |
| ldi; ldd | ? | ? | 0 | ! | 0 | – |
| cpi; cpir; cpd; cpdr | ! | ! | ? | ! | 1 | – |
| bit b,s | ? | ! | 1 | ? | 0 | – |

Zeichenerklärung:
! Flag wird entsprechend beeinflußt
? Flag wird willkürlich beeinflußt
1 Flag wird gesetzt
0 Flag wird zurückgesetzt
– Flag wird nicht beeinflußt